DESIGN AND IMPLEMNTATION OF AN EDUCATIONAL AM RECEIVER WITH FPGA USING SDR TECHNIQUES

Ali Ibrahim Khalifa HANDER

2021
MASTER THESIS
ELECTRICAL AND ELECTRONICS ENGINEERING

Thesis Advisor
Assist. Prof. Dr. Bilgehan ERKAL

DESIGN AND IMPLEMNTATION OF AN EDUCATIONAL AM RECEIVER WITH FPGA USING SDR TECHNIQUES

Ali Ibrahim Khalifa HANDER

T.C.
Karabuk University
Institute of Graduate Programs
Department of
Electrical and Electronic Engineering
Prepared as Master Thesis

Thesis Advisor
Assist. Prof. Dr. Bilgehan ERKAL

KARABUK
January 2021

I certify that in my opinion the thesis submitted by Ali Ibrahim Khalifa HANDER titled "DESIGN AND IMPLEMNTATION OF AN EDUCATIONAL AM RECEIVER WITH FPGA USING SDR TECHNIQUES" is fully adequate in scope and in quality as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Bilgehan ERKAL ……
Thesis Advisor, Department of Electrical and Electronic Engineering

This thesis is accepted by the examining committee with a unanimous vote in the Department of Electrical and Electronic Engineering as a master thesis. January 19, 2021

Examining Committee Members (Institutions)	Signature
Chairman: Assoc. Prof. Dr. Salih GÖRGÜNOĞLU (KU)	……
Member : Assoc. Prof. Dr. Hüseyin DEMIREL (KBU)	……
Member : Assist. Prof. Dr. Bilgehan ERKAL (KBU)	……

The degree of Master of Science by the thesis submitted is approved by the Administrative Board of the Institute of Graduate Programs, Karabuk University.

Prof. Dr. Hasan SOLMAZ ……
Director of the Institute of Graduate Programs

"I declare that all the information within this thesis has been gathered and presented in accordance with academic regulations and ethical principles and I have according to the requirements of these regulations and principles cited all those which do not originate in this work as well."

Ali Ibrahim Khalifa HANDER

ABSTRACT

M. Sc. Thesis

DESIGN AND IMPLEMNTATION OF AN EDUCATIONAL AM RECEIVER WITH FPGA USING SDR TECHNIQUES

Ali Ibrahim Khalifa HANDER

Karabük University
Institute of Graduate Programs
Department of Electrical and Electronic Engineering

Thesis Advisor:
Assist. Prof. Dr. Bilgehan ERKAL
January 2021, 102 pages

In this study, an AM receiver is designed and implemented in FPGA using SDR techniques. The main purpose of the study is to provide a cheap and simple FPGA based platform for education of SDR basics. Firstly in the study, a simulation environment is set using MATLAB scripts. A set of test signals are recorded and used to generate an AM test signal using MATLAB scripts. The signal is used in simulation and test of FPGA implementation. Simulation code is also used as a framework in the VHDL design of the FPGA based SDR system. Another MATLAB script is written to analyze the test and simulation results and make a comparison. Where the results obtained from those tests on the two signals, it proved that the tests with the signal A1 are better than the tests with the signal A2, as the higher SNR ratio means the better. When comparing the actual real-world values with the simulations for each test signal, it is noted that the real-world SNR results are slightly lower than the simulations SNR. Where test results provided a SNR higher than

20dB, which is an acceptable level for an AM receiver. Where, test and simulation results prove FPGA AM RX system a useful candidate for AM demodulation and reception. The designed and implemented FPGA AM RX system is also a good utility in the education of basic SDR principles which is the main focus of this study.

Key Words : SDR, FPGA, VHDL, SNR, MATLAB, RX, AM.
Science Code : 90523

ÖZET

Yüksek Lisans Tezi

SDR TEKNİKLERİNİ KULLANARAK FPGA İLE EĞİTİM AMAÇLI BİR ALICININ TASARIMI VE UYGULANMASI

Ali Ibrahim Khalifa HANDER

Karabük Üniversitesi
Lisansüstü Eğitim Enstitüsü
Elektrik-Elektronik Mühendisliği Anabilim Dalı

Tez Danışmanı:
Dr. Öğr. Üyesi Bilgehan ERKAL
Ocak 2021, 102 sayfa

Bu çalışmada, bir AM alıcısı, SDR teknikleri kullanılarak FPGA'da tasarlanmış ve uygulanmıştır. Çalışmanın temel amacı, SDR temellerinin eğitimi için ucuz ve basit bir FPGA tabanlı platform sağlamaktır. İlk olarak çalışmada MATLAB betikleri kullanılarak bir simülasyon ortamı oluşturulmuştur. MATLAB komut dosyalarını kullanarak bir AM test sinyali oluşturmak için bir dizi test sinyali kaydedilir ve kullanılır. Sinyal, FPGA uygulamasının simülasyonunda ve testinde kullanılır. Simülasyon kodu, FPGA tabanlı SDR sisteminin VHDL tasarımında bir çerçeve olarak da kullanılır. Test ve simülasyon sonuçlarını analiz etmek ve bir karşılaştırma yapmak için başka bir MATLAB betiği yazılmıştır. İki sinyal üzerinde yapılan bu testlerden elde edilen sonuçlar, daha yüksek SNR oranı daha iyi anlamına geldiğinden, sinyal A1 ile yapılan testlerin A2 sinyali ile yapılan testlerden daha iyi olduğunu kanıtlamıştır. Gerçek gerçek dünya değerleri her bir test sinyali için simülasyonlarla karşılaştırılırken, gerçek dünya SNR sonuçlarının simülasyon

SNR'sinden biraz daha düşük olduğu not edilir. Test sonuçlarının, bir AM alıcısı için kabul edilebilir bir düzey olan 20dB'den daha yüksek bir SNR sağlaması durumunda. Test ve simülasyon sonuçları, FPGA AM RX sisteminin AM demodülasyonu ve alımı için yararlı bir aday olduğunu kanıtladı. Tasarlanan ve uygulanan FPGA AM RX sistemi, bu çalışmanın ana odak noktası olan temel SDR ilkelerinin eğitiminde de iyi bir yardımcı programdır.

Anahtar Kelimeler : SDR, FPGA, VHDL, SNR, MATLAB, RX, AM.
Bilim Kodu **:** 90523

ACKNOWLEDGMENT

I would like to express my appreciation to my great supervisor, Assist. Prof. Dr. Bilgehan ERKAL who has given me an unlimited support and valuable guidance. There are no enough words to express thanks to him.

As well as, I would like to thank my lovely family from my heart for their being with me by supporting me with all possible means.

CONTENTS

Page

Page

Page

LIST OF FIGURES

Page

Page

LIST OF TABLES

Page

SYMBOLS AND ABBREVITIONS INDEX

ABBREVITIONS

AM : Amplitude Modulation
FM : Frequency Modulation
SDR : Software Defined Radio
DSP : Digital Signal Processors
FPGA : Field Programmable Gate Arrays
RF : Radio Frequency
IF : Intermediate Frequency
DSB : Double Side Band
DUC : Dynamic Update Client
DUC : Digital Up Conversion.
DAC : Digital-to-Analog Converter
PLL : Phased Locked Loop
ADC : Analog to Digital Converter
ASIC : Application Specific Integrated Circuit
DDC : Digital Down Converter
HDL : Hardware Description Language
LUT : Look Up Table
RAM : Random Access Memory
VHSIC : Very High Speed Integrated Circuit
VHDL : VHSIC Hardware Description Language
PCB : Printed Circuit Board
FF : Flip-Flop
IC : Integrated Circuit
PLD : Programmable Logic Device
CLB : Configurable Logic Blocks
RTL : Register Transfer-Level

CHAPTER 1

INTRODUCTION

Since the emergence of cellular communication in the last two decades, wireless communication channels gained more popularity. Currently, the development of wireless applications and wireless technology needs flexibility in the hardware. It is time-consuming and so expensive to make new radios in response to the change of wireless applications and standards [1]. These problems can be solved through software radio by moving the components of analogs to the digital domain. Functions of radio can be implemented by using programmable logic devices including Field Programmable Gate Arrays (FPGAs).

FPGAs offer the ability to implement functions in a cheap way which were previously implemented by using the components of analog hardware. They are constructed by one basic reconfigurable logic cell that doubled thousands of times. FPGAs are used as co-processors to interact with DSPs and general-purpose processors and offering lower cost and higher performance to the system. The freedom to select where to implement the baseband-processing algorithms will add another flexibility dimension when using Software Defined Radio (SDR) algorithms. Therefore, with SDR, it is possible to implement simply the radio communitarian process. We can say that SDR is better than the conventional radio communication system because with SDR all hardware is removed and replaced by pure software.

Moreover, this flexibility gives an advantage to SDR receiver where it will be able to decode the entire signals.

Besides, software radio permits a single device on receiving numerous and different wireless transmissions. By the use of digital signal processing mechanisms of FPGAs, the software radio may be accomplished in the digital systems. Nevertheless,

it is logical to focus on AM transmission rather than FM because constructing AM receiver is very easy to be learned. This technique is beneficial because it develops digital design mechanisms that can be applied in more advanced communication systems. Besides, implementing AM receiver by the use of analog electronics is always the base. Nevertheless, the development of digital systems allows the emulation of analog with digital circuity easily. The digital AM Receiver is a digital system that tries to accomplish the same analog AM receiver functions by using only FPGA and a small number of analog electronics.

In general, AM is a technique that is used to modulate the wave based on changing its amplitude based on changes in the frequency and amplitude together of the associated modulating signal and keep the frequency of the wave constant. The change in wave amplitude is directly associated with the change in the modulating signal amplitude. The changes in the modulating signal determine the positive and negative peaks of the wave change. Increase or decrease the modulating signal amplitude causes an associated increase or decrease of the wave peaks amplitude [2].

1.1. LITERATURE REVIEW

In 1894-5, Marconi, Oliver Lodge and Alexander Popov invented the first radio receiver by the use of a primitive radio wave detector named as a cohered and invented in 1890 by Eduard Branly and improved by Lodge and Marconi [3]. At the beginning, it has high resistance. When the voltage of radio frequency was applied to the electrodes, its conducted electricity and resistance has been reduced. The coherer in the receiver was directly connected between the ground and antenna. Moreover, the coherer was connected to a DC circuit with a relay and battery besides its connection to the antenna. When the coherer resistance is reduced by the incoming radio wave, the current of battery flowed through it turning on the relay to ring a bell or create a mark on the tape of paper in the siphon recorder.

For restoring the coherer to the previous no conducting status in order to receive the next pulse of radio waves, it had to be tapped mechanically to disturb the metal particles [4]. This has been performed by a "DE coherer", and it is a clapper struck

the tube and operate by the electromagnet powered by a relay. In 1970, a researcher coined the term of “digital receiver”. The Gold Room Laboratory in California generated an analysis of software baseband tool named Midas that was of course a software defined. In 1984, a team in Garland, Texas Division of E-Systems was coined the term “Software Radio”. The same place witnessed the development of the 'Software Radio Proof-of-Concept' laboratory which published the software radio inside many governmental organizations. This software radio which has been invented in 1984 was a digital baseband receiver which offered programmable interference cancellation and demodulation for broadband signals, typically with thousands of adaptive filter taps by the use of many array processors retrieving shared memory. In 1991, the term ‘software radio’ has been reinvented by Joe Mitola independently in a plane to construct a GSM base station which may combine between Ferdensi's digital receiver with E-Systems Melpar's which control digitally the communications jammers to a true software-based transceiver.

The first main push in SDRs development is implemented by the use US military paper called SpeakEasy. The main objective for SpeakEasy paper was to use the programmable processing to simulate more than 10 presenting military radios which operate in frequency bands between 2 and 2000 MHz. The other design goal was to be able to easily include a new modulation and coding criteria in future. Therefore, the military communication can keep pace with the developed modulation and coding techniques. From 1992 to 1995, the main goal was to create a radio for the U.S. Army which would operate from 2 MHz to 2 GHz with satellites, Naval Radios, Air Force radios and ground force radios.

The main goal was to obtain a quicker reconfigurable architecture at open software architecture with cross channel connectivity (the radio may bridge various protocols of radio).

Harnani Hassan et al [5], from University Teknologi MARA implemented a low complexity SDR using Simulink, Matlab and Xilinx environment based on FPGA. Xilinx has been used as a platform used as a method for FPGA design whereas the Matlab and Simulink has been used to create a random spectrum signal. The

methodology and mechanisms of the proposed transceiver design helped to design the SDR by offering a quick method altering system with low complexity. As well as, it opened the way to integrate cognitive radio aspect to wireless network including 3G and 4G in the future. As well as, the proposed design accomplished the goal and proved that it can be easily conducted by the use of Blockset, Xilinx DSP and Simulink. This design method provide a benefit to designers on using either HDL, Verilog or Matlab. As well as, it helped designers to determine the problems and provide a quick method to alter the system.

Shahana K et al [6], designed and implementation of Low Frequency Trans-Receiver on Spartan-3AN device. System Generator has been used to design and to simulate system level models, and to get the timing and resources using results before conducting the design on actual device. The primary idea behind this is to seek about the feasibility to get the software close to the antenna as can as possible and therefore, solve the problems of hardware by using software. The benefit of this method is that the equipment is relatively cheap and more versatile. Also, the low frequency trans-receiver based on FPGA is simple to be upgraded and offers high flexibility in execution. The measured findings shown that the input of transmitter matches with the output of receiver. Furthermore, Simulation of Matlab has been implemented for further aware to the mentioned problem. The comprehensive implementation is considered a perfect example to conduct the problem of hardware in software. In addition, it offers low power solution and low cost. FPGA implementation may further deliver flexibility to customize the design on different data ratios, Carrier Frequency, Filter types, Modulation types, etc. which make the design efficiently reconfigurable.

Jiang-tao Gong et al [7], from Hunan Railway Professional Technology College presented the block diagram for FPGA to realize the distributed algorithm which can implement the SDR channel processing, where it consists a multiple FIR filters bank for various frequency bands because the radio system defined by a software need into a series of different FIR filters to catch the equivalent signal. Through the distributed algorithm depending on signal processing structure of FPGA and by Repeating configure the FPGA, it may accomplish more FIR filter bank switching, in order to

accomplish different channel information receiving. It offers applicable processing approaches and thoughts for radio channel switching defined by software.

In this study, an AM receiver using SDR techniques is designed and implemented in FPGA on Spartan 6 FPGA Board with LM4550 Audio and IO Breakout Board. The basic notion behind is to seek provide a cheap and simple FPGA based platform for teaching and learning of SDR basics. Where a simulation environment is set using MATLAB scripts. And a set of test signals are recorded and used to generate an AM test signal using MATLAB scripts. The signal is used in simulation and test of FPGA implementation. Simulation code is also used as a framework in the VHDL design of the FPGA based SDR system. Another MATLAB script is written to analyze the test and simulation results and make a comparison.

Test and simulation results prove FPGA AM RX system a useful candidate for AM demodulation and reception. Subsequently the designed FPGA AM RX system a good in the education of basic SDR principles. Also it can be used in teaching the radio signal processing techniques using FPGAs. The system is also suitable to be used with any soundcard based SDR frontend.

CHAPTER 2

SOFTWARE DEFINED RADIO (SDR)

Radio development in the communication field which people need, comprising video and voice communication and broadcasting messages, etc. Radio SRD is the definition of system software that comprises the entire or many descriptors including modification, extraction, and others. Wireless devices are used easily and cost less business mission. Software-defined radio (SDR) provides many advantages where it pushes forward the cost of communication and flexibility with several advantages accomplished by the service providers to the end-users. You can obtain more than an explanation for the software-defined Radio also called (SDR). Radio is a wireless device that sends information and receives frequencies. Many issues must be solved in order to access SDR including tuning the specification of the system according to numerous applications. We may return some of all will be held including modification, extraction and encoding. To end it, this information helps the recognition of these specifications through the reception [8].

2.1. SDR ADVANTAGES

- Point and Click Control
- Easy Tuning
- A Computer Is Sharing the Workload
- Cheaper (In Some Cases)
- Smaller
- Visual look at a signal
- Open Platforms
- Custom Filtering Uses modern technology

2.2. SDR DISADVANTAGES

- Filtering Traded For Space
- Hard to run on old computers
- Sending is more expensive
- Dependent on Computer
- Software Limits

2.3. IDEAL SDR DESIGN

Software-defined radio system (SDR) is considered one of the most significant contemporary techniques in supporting the communication in military service insecurity, war and peace. SDR is used rather than the conventional radio and it involves optimal radio frequency RF convertor wireless signal to an analog IF properly used in conventional radio. Analog signal conversion to a digital frequency (ADC) in IF and convert the signal from digital to analog FM frequency in the IF is called (DAC) and shown in Figure 2.1.

The transfer of the signal routed by the converter sample rate by the interface (ADC) and the treatment of hardware in the receiver. SDR may use the processing of baseband with several digital devices including digital signal processors (DSP) and field-programmable gate arrays (FPGA).

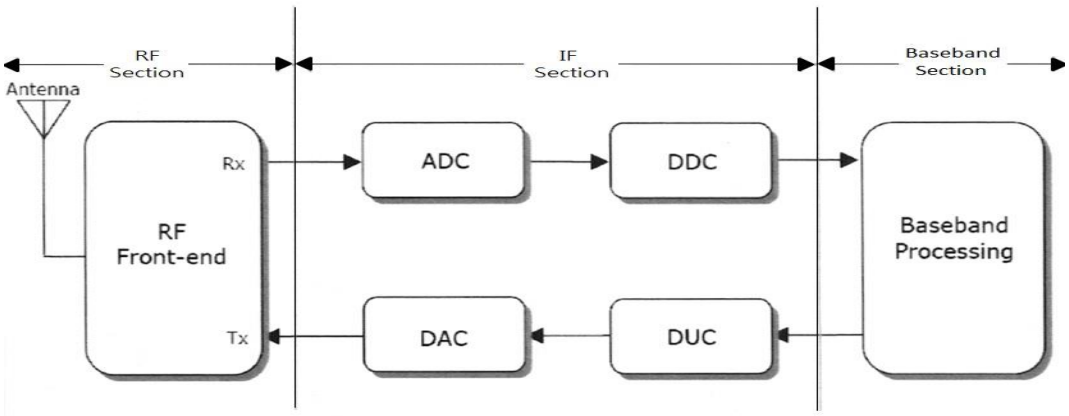


Figure 2.1. Functional block diagram of wireless communication system.

There are many advantages to the use of digital devices such as low energy consumption, high processor speed and flexibility. Nevertheless, there is a comparison between the extreme flexibility degrees with the increase in consumption of energy for DSP to minimum limit the flexibility and lower consumption of energy than ASICs. FPGA provides astray consistent devices that cheap and less energy consumption than DSP and ASICs flexibility FPGA and the redesign is made it optimal of SDR [9].

2.4. MOTIVATION AND OBJECTIVES

SDR looks like many technologies in terms of its development and it is used in both military and civilian applications and called Speakeasy. It is used in the naval forces of the United Stated between 1991-1995. This technology accomplished great success in the basic rules, knowledge, radio program, wireless communication and programming. Currently, all SDR software is available at low prices [10].

2.5. SDR HARDWARE

2.5.1. Traditional Receiver

In addition to the classic demodulation, the traditional receiver and the three processes to determine the sign in the carrier frequency setting of frequency shifting, the candidate is filtering or separated from others. The compensation for transport losses by enlargement is inserted enlargement by mass demodulation. Because of carrying the signal to the demanded level circuit demodulation, most of the conventional reception setups use different plans for about a century. Figure 2.2 shows the basic structure is significant to differentiate between the conventional and reception by new SDR methods.

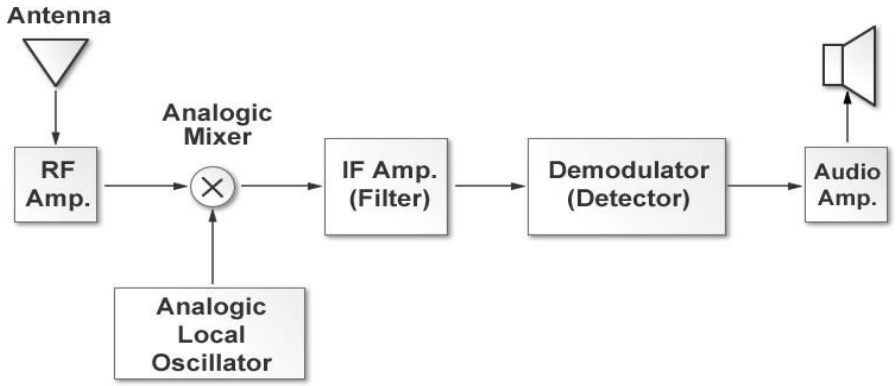


Figure 2.2. Internal blocks of super heterodyne receiver.

Figure 2.2 clarifies the signal interference by the antenna. The signal is amplified during RF phase that works in the frequency area of benefit only after passing the reference to the mixer during the other input which receives the contribution oscillator ornament and appointing local frequency oscillator. By tuning the radio which is responsible to translate the frequency signal mediator (IF), the mixer is responsible for shifting the frequency to the medium frequency IF.

The purpose of appointing a frequency oscillator is to confirm that the amount of time difference frequency signal is equal to (IF). For instance, if the frequency at the FM station is 100.7 MHz and IF was rumored to 10.7 MHz, the oscillator should be adjusted to 90 MHz situation due to the low side transformation. The following phase is the phase of weakening all the candidate wave signal but certainly a part of the spectrum. The received signal of the band is prevented to display by the bandwidth. At the end of the stage, the original signal modified is restored by the demodulator through the loudspeaker IF it uses one substitute. To increase the processing of the signal, it depends on the purpose through which its intended recipient device. Cross of information learned to a loudspeaker connected to the speaker [11, 12].

2.6. SDR RECEIVER

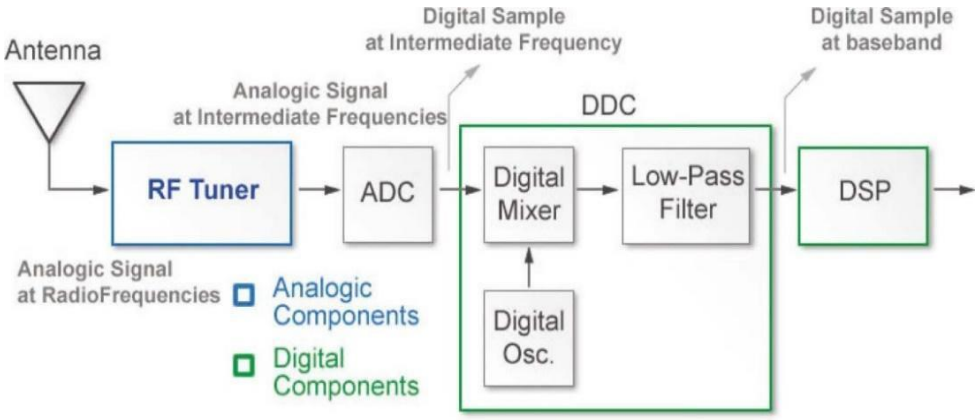


Figure 2.3. Block diagram of the SDR receiver.

Figure 2.3 for receiver signal SDR and be the first fund pass it RF tuner transfers the analog signal to IF to be the same and conduct the process in the first three boxes of the variant receiver device to the convergence point of the two systems [13].

Then cross-reference IF are changed frequency band by ADC is liable on change and is fed into the next phase and be down the digital converter. DDC is a significant part of the SDR system, it is cheap and consists of three main parts as follows:

- Digital local oscillator.
- Digital mixer.
- Finite pulsation response FIR low-scrolling IF filter.

Reference transfers to the corresponding baseband in our digital mixer at the counter of phase elements by the analysis [14]. It is a modified digital local oscillator that the reference is needed far or up to 0Hz and the difference with the bandwidth along and be a low-pass filter and detects any receiver part is a suitable signal. Another approach is represented by decreasing the sampling ratio or sampling frequency is taken to new samples from the baseband and create from the split in the frequency of the original sample through an N element. It is named the decimation element. The ratio of the end sample may be less than double the higher-frequency elements by Nyquist theory.

The samples crossing to the baseband digital signal processing in a DSP box, finally, for example decoding and demodulating [15].

2.7. SDR TRANSMITTER

As shown in Figure 2.4, DSP income generates the baseband signal to be sent by SDR. The first box is DUC for digital transformation and translates the baseband signal for IF by make its passband.

DAC send the samples to the field analog after the RF is moving towards the high-frequency signal is later enlarged and the signal transmitted from the antenna DUC Filter is responsible for the high sample ratio of the baseband signal which is compatible with the operating of the elements followed by the so reverse process arises at the reception frequency [14, 15].

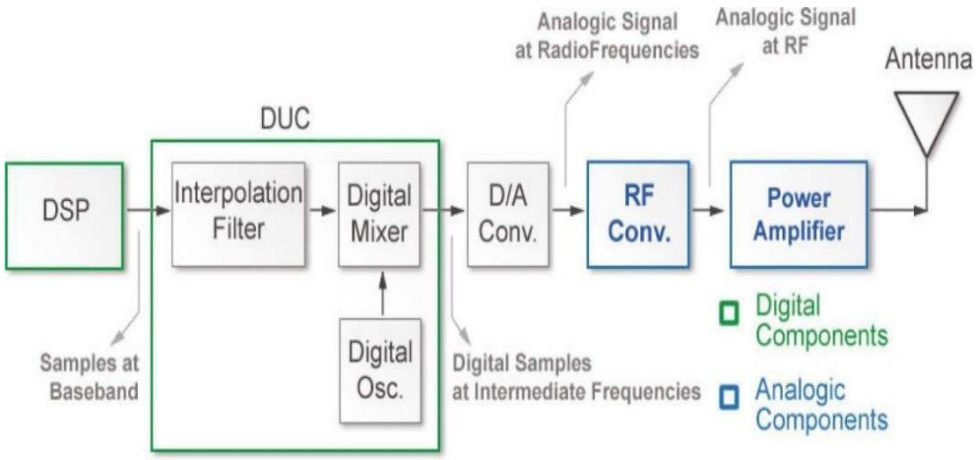


Figure 2.4. Block diagram of a SDR transmitter.

CHAPTER 3

FIELD PROGRAMMABLE GATE ARRAYS (FPGA)

FPGA can be described as a device that includes a matrix of reconfigurable gate array logic circuitry. When FPGA is formed, the internal circuitry is connected in a method that produces a hardware implementation of the software application. FPGAs do not include an operating system and they use dedicated hardware to process logic. The nature of FPGAs is parallel and therefore, different operations are not competing for the same resources. Consequently, when adding additional processing, one part of the application performance is not influenced.

Besides, many control loops can operate on a single FPGA device at different ratios. The critical interlock logic can be enforced by FPGA-based control systems and can be designed to inhibit I/O enforced by the operator. Nevertheless, FPGA-based systems are unlike the hard-wired printed circuit board designs that have stable hardware resources where FPGA-based systems may rewire their inner circuitry to help the reconfiguration when the control system deploys in the field.

FPGA offers the reliability and performance for the dedicated hardware circuity. By the use of FPGA, it is possible to substitute thousands of discrete elements by merging millions of logic gates in one integrated circuit (IC) chip. As shown in Figure 3.1 the internal resources of FPGA chip include a matrix of configurable logic blocks bounded by a periphery of I/O blocks. Inside the FPGA matrix, the signals are routed by wire routes and programmable interconnect switches.

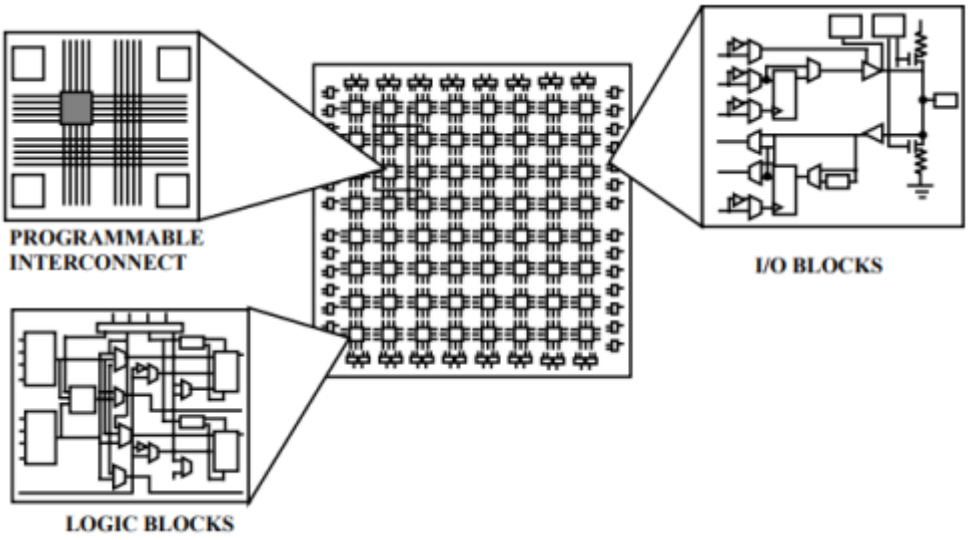


Figure 3.1. Internal structure of FPGA.

3.1. FPGA ARCHITECTURE

The FPGA structure consists of many components as follows:

- Look-up table (LUT): This component conducts many logical operations.
- Flip-Flop (FF): This register component stores LUT result.
- Wires: These components connect components.
- Input/output (I/O) pads: These ports are physical and their mission is to get data in and out of FPGA.

The collection of these components produce the basic structure of FPGA as clarified in Figure 3.2. Despite the efficiency of this structure to implement any algorithm, the proficiency of the resulting implementation is limited in terms of calculated output, feasible clock frequency and demanded resources.

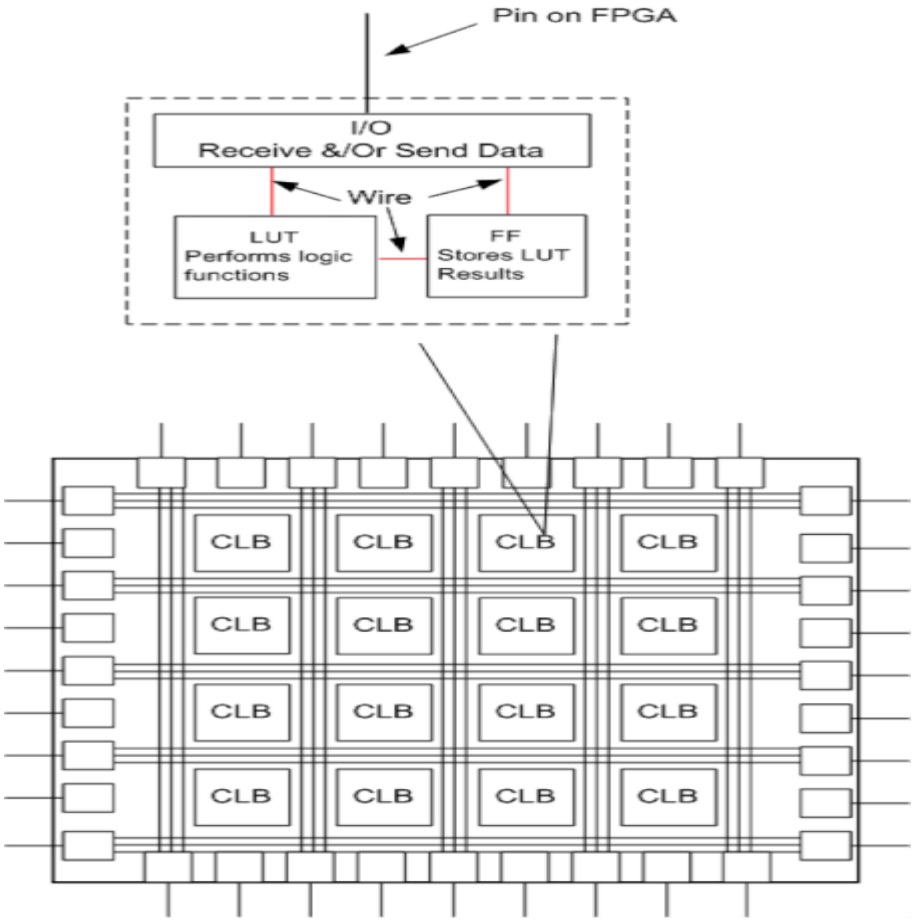


Figure 3.2. Basic FPGA architecture.

The modern architecture of FPGA includes many basic components accompanied by other computational and storage blocks which increase the effectiveness and computational density of the device. The additional components which will be discussed in the following sections are as follows:

- Embedded memories to store the distributed data.
- Phase-locked loops (PLLs) to drive the FPGA fabric at different clock ratios.
- High-speed serial transmitting and receiving devices.
- Off-chip memory controllers

The collection of these components gives FPGA the flexibility in implementing any software algorithm running on processors and produce the modern FPGA architecture shown in Figure 3.3.

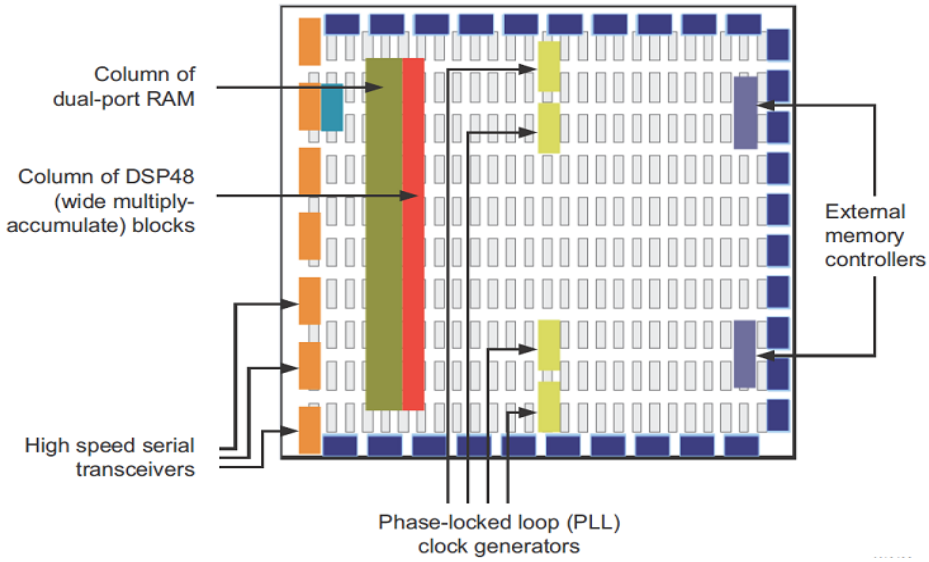


Figure 3.3. Contemporary FPGA architecture.

3.1.1. Logic Cells

The simple FPGA includes a large number of logical cells and each cell can be configured to conduct many functions. Each logic cell has a unified number of entries and exits. The logic cells used in FPGAs are as follows:

- Multiplexer based logic cells (e.g. Actel FPGAs)
- Memory-based logic cells (e.g. Xilinx FPGAs)

The basic internal structure of FPGA in a very wide sense is shown in Figure 3.4.

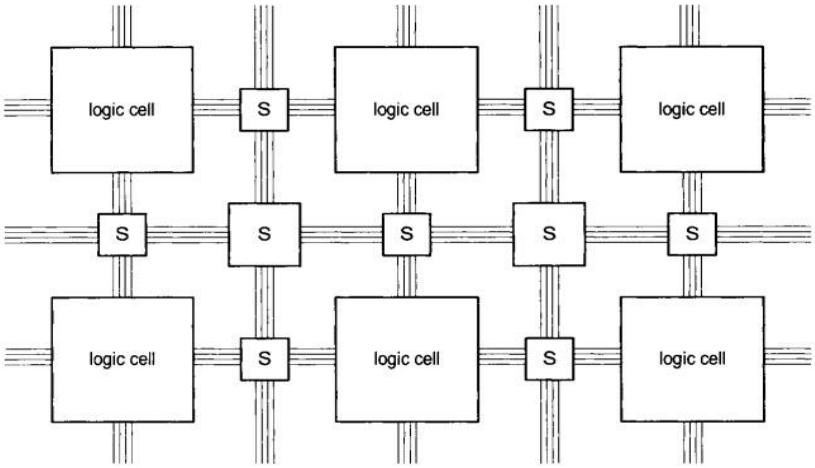


Figure 3.4. Logic cells.

As shown in Figure 3.4 that the internal structure of FPGA consists of programmable interconnections and configurable logic cells.

CHAPTER 4

AMPLITUDE MODULATION (AM)

The amplitude modulation occurs when high-frequency carrier wave amplitude differs as a function of signal intensity. Figure 4.1 shows the principle of amplitude modulation. We can realize that the amplitudes of positive and negative carrier wave half-cycles differ in relation to the signal. This means that increasing the positive sense results lead to an increase in the carrier wave amplitude whereas the opposed happens for the negative half-cycle. In general, AM process is implemented by the use of an electronic circuit which is called a modulator [16].

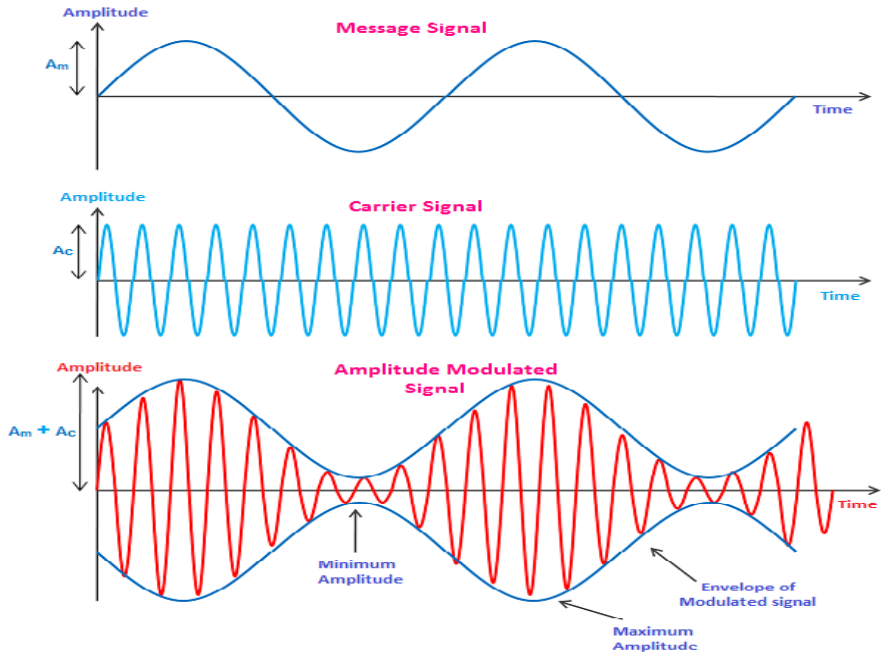


Figure 4.1. Representation of the AM principle.

The AM process includes an important consideration which is the modulation factor. We can imagine this factor as the depth of modulation or change in carrier amplitude. In other words, it represents the ratio between the change in carrier amplitude and the

amplitude of normal carrier wave. The purpose of this factor is the determination of strength and quality of transmitted signals respectively. The modulation of the carrier in AM wave to a small degree produces a small change in carrier amplitude. Therefore, the transmitted audio signal is not too strong. This means that the audio signal is stronger and clearer when the modulation degree is great [16].

4.1. DOUBLE SIDE BAND AMPLITUDE MODULATION (DSB-AM)

One of the main types of AM technique is Double Side Band (DSB) where DSB consists of two sidebands upper and lower with wave carrier suppressed. Practically, DSB is consistent with SSB receivers where the last one is considered of the main types of AM techniques in which the receiver rejects only the unwanted or redundant sideband.

DSB signals are generated based on suppressing the carrier that results in the upper and lower sideband. This generating method does not consist of waste in power.

DSB signal is generated based on modulating a carrier across the information signal of a single-tone sine wave and signifies the summation of two sinusoidal sidebands. Later, the carrier suppressed and the amplitude of the DSB sine wave signal changes in the frequency of the carrier. The main features of DSB signal are the transition of the stage that occurs at the wave lower amplitude slices. In general, DSB carrier signals are generated by the balanced modulator circuit based on generating the difference or summation between frequencies and to cancel or balance the carrier. However, DSB signals are rarely used despite these features and both the low cost and simple design because it is difficult to demodulate signals at the receiver. DSB signals are used in many applications but the most important one is the transmitting of information in television signals [17].

4.2. DSB-AM RECEIVERS

One of the oldest radio modulation technique is the amplitude modulation. The receivers which are used to listen to DSB-AM are maybe the simplest receivers for

any radio modulation technique that perhaps the reasons behind the use of this versions of amplitude modulations until now. The super heterodyne type of receivers is the most popular receivers in use currently. They comprise of Amplifier, Local Oscillator and Mixer, IF Section, Antenna, and Detector RF amplifier. The need of these systems can be noticed when we consider the simplest and inadequate TRF or tuned radio frequency amplifier. Amplitude modulation happens when the carrier wave amplitude is modulated in order to respond to the source signal. In amplitude modulator, we have an equation which is look this:

$$A_{signal}(t) = A(t)\sin(\omega t) \tag{4.1}$$

As where the much simple form amplitude modulation modulator comprises of a diode that is configured to represent a detector of envelop. Product detector is considered another type of demodulator which provide better-quality demodulation with further circuit complexity.

4.2.1. Modulation Spectrum

As treated previously, the beneficial modulation signal *m(t)* is frequently more complicated than a single sine wave. Nevertheless, in accordance with Fourier decomposition, *m(t)* can be convoyed as the set of sine wave sum for many stages, frequencies and amplitudes.

By performing the multiplication of *1 + m(t)* with *c(t)* as previously, the result comprises a sum of sine waves. The carrier *c(t)* presents unchanged, but every frequency element of *m* at f_i has two sidebands at frequencies $f_c + f_i$ and $f_c - f_i$. The set of previous frequencies above the carrier frequency is called as upper sideband and those lower configure the lower sideband. As clarified in upper of Figure 2, the modulation *m(t)* can be considered to comprise an equal combination of positive and negative frequency factors. The sideband can be viewed as the modulation *m(t)* which is simply shifted in frequency by f_c as showed at the bottom from the right of Figure 2.

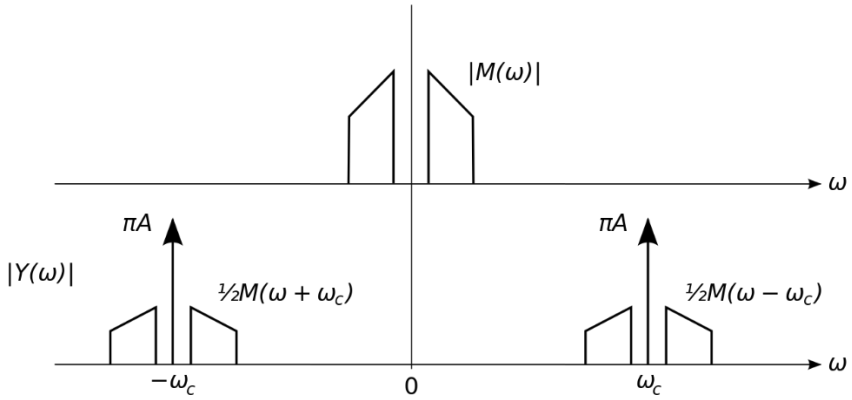


Figure 4.2. Spectra of double-sided for AM signals and baseband.

4.2.2. Demodulation Methods

The simplest type of amplitude modulation comprises a diode that is configured to represent an envelope detector. Product detector is considered other type of demodulator that has the ability to offer better-quality demodulation with further circuit complexity.

4.2.2.1. Envelope Detector

When there is an attempt to demodulate the modulation amplitude, it looks like a good sense which only the amplitude of the signal need to be cheeked. By checking only the amplitude of signal at specific time, it is possible to eliminate the carrier signal from consideration and it is possible to check the original signal. The amplitude of signal can be checked by using a tool in our toolbox (the envelop detector).

The envelope detector is just a half wave rectifier followed by a low pass filter. It can be imagine as electronic circuit which takes (comparatively) high-frequency amplitude modulated signal as input and delivers an output that is the demodulated envelop of the original signal as shown in Figure 4.3.

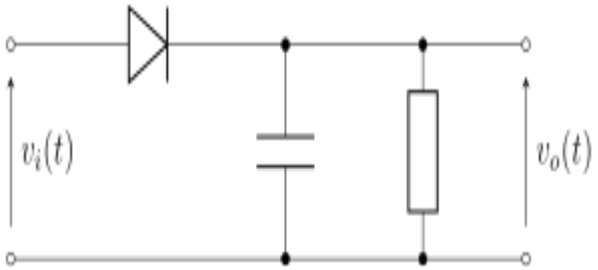


Figure 4.3. The simple circuit of envelope demodulator

The detector is placed after the IF section in case of commercial amplitude modulator. In this point, the carrier is 455 kHz whereas the highest frequency of envelop is only 5 kHz. Since the ripple element is about 100 times the frequency of the maximum baseband signal and does not pass through any succeeding audio amplifiers. We can see below the forms of AM or FM signal $x(t)$ as follows:

$$x(t) = R(t)\cos(\omega t + \varphi(t)) \tag{4.2}$$

In the case of AM, $\varphi(t)$ (the stage element of the signal) is constant and can be neglected. Moreover, the carrier frequency ω of amplitude modulator is also constant. Therefore, the entire information of the amplitude modulator signal is in $R(t)$ where $R(t)$ is known as the envelope of the signal. Thus, the amplitude modulator signal is given by the function as follows:

$$x(t) = \big(C + (mt)\big)\cos(wt) \tag{4.3}$$

C represents the carrier amplitude, (mt) represents the original audio frequency message and $R(t)$ equal to $C + (mt)$. Consequently, the original message can be recovered if the envelop of the amplitude modulator is extracted.

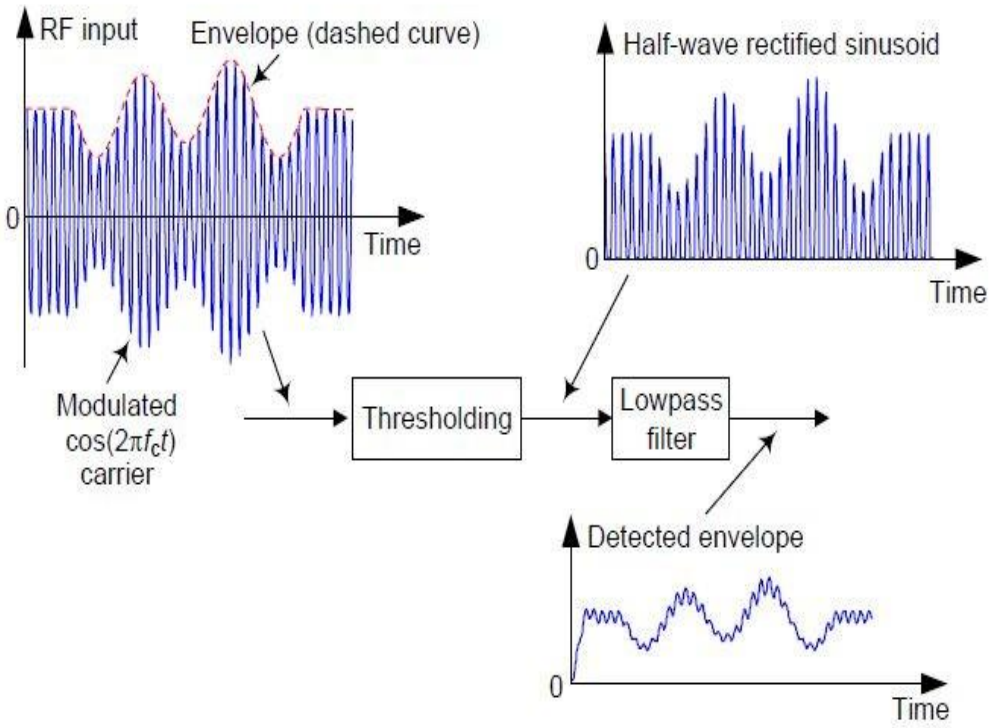


Figure 4.4. A Signal and its envelope detector.

4.2.2.2. Square Law Detector

A square low detector of electronic signal processing is a device which create an output relative to square of some input. For instance, in radio signal demodulation, semiconductor diode is used as a square law detector which provide an output existing relative to the square of the amplitude of the input voltage over some range of input amplitudes. A square law detector offers an output directly relative to the power of the input electrical signal. Moreover, square detector is a coherent or synchronous detector. It avoids the problem to recreate the carrier by simply square the signal of input. Basically, it uses the amplitude modulator signal itself as a range of wideband carrier. The multiplier output is the square of the input amplitude modulator signal:

$$(e_{am})^2 = (\sin \omega_c t + \frac{m}{2} \cos(\omega_c + \omega_m)\, t - \frac{m}{2}\, cos(\omega_c + \omega_m)t)^2 \qquad (4.4)$$

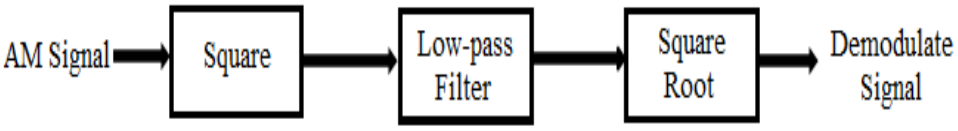


Figure 4.5. Block diagram for squaring law.

CHAPTER 5

VHDL –HARDWARE DESCRIPTION LANGUAGE

VHDL is a term that refers to the Very high speed integrated circuit (VHSIC) Hardware Description Language (VHSIC). VHDL is a programming language used to describe the logic circuit by functioning the behavior of data flow and/or the structure.

The description of this hardware is used to configure a programmable logic device (PLD) including a Field Programmable Gate Array (FPGA) with a convention logic design. Also, VHDL is a formal language used to specify the structure and behavior of the digital circuit.

5.1. VHDL CONCEPTS

The goal of VHDL is to describe a model of digital hardware device where this model identifies the external view of the device and one or more of the inner views. The external view of the device identifies the interface of the device which it can communicate with many models in the same environment while the internal view of the device identifies the structure or functionality.

5.1.1. Behavoaral Modelling

The most basic formula for the behavioral modeling of VHDL is the signal assessment statement as shown in the following example:

```
a <= b;
```

The previous example means that a gets the value of b. This statement has an effect which is signal a is replaced by signal b. When the value of signal b is changed, this

statement is executed. The sensitivity list of this statement is signal b. The signal statement is executed when a signal in the sensitivity list of a signal assignment statement changes value. If the execution produced a new value that is different from the present value of the signal, the event will be scheduled for the target signal. Consequently, no event will be scheduled but the transaction is still generated if the execution result value is the same. The event is scheduled by only the changes of the value while the transaction is always generated when the model is assessed.

A transaction is always generated when a model is evaluated, but only signal value changes cause events to be scheduled.

5.1.2. Structural Modelling

The structural description describes the logical elements of the system and thus, it is a simulation of the system. The elements can be OR gate(s), AND gate(s), or it can be at a higher logical level for example Processor Level or Register Transfer-Level (RTL). The structural description is more traditionally used than the behavioral description for the system which requires explicit design. If we want to operate A + B=C. In behavioral design, we must write C = A + B and we have no choice on the type of adders used to conduct this addition process.

The entire statements with the structural description are concurrent. The entire statements that have an event conducted concurrently through any simulation time.

The main difference between VHDL and Verilog structural description is the availability elements (particularly primitive gates) for the user. The whole primitive gates including AND, OR, XOR, NOT and XNOR are recognized in Verilog. The gates must be linked to library, packages or modules which have the description of gates to be recognized by the VHDL packages.

5.1.3. RTL (Register Transfer Level) Diagrams

Register-transfer level (RTL) that existed in the digital circuit design is modeling the asynchronous digital circuit in terms of digital signals flow between the logical operations conducted on those signals and hardware registers. Register-transfer level abstraction is used hardware description languages (HDLs) such as Verilog and VHDL to generate high-level illustrations of a circuit through which lower-level representations and at the end actual wiring are derived. RTL design is considered a distinctive practice for modern digital design [18].

In contrast with software complier design when register-transfer level intermediate exemplification is the lowest level, RTL level is the ordinary input which designers of circuit operate on there are various more level than it. Actually, in the synthesis of the circuit, a transitional language is used between input register transfer level representation and target netlist. Unlike in netlist, constructs for example cells, functions and multi-bit registers are existed [18].

5.2. VHDL DESIGN STAGES

5.2.1. Entity

Entire designs are expressed in terms of entities. The most basic building block of design is the entity. The entity of VHDL determines the entity name, entity ports and information related to the entity. Entire designs are created by the use of one or more entities [19]. If the type of design is hierarchal, the description of the top-level will include a description of the lower-level contained in it.

5.2.2. Architecture

The architecture description is included in all entities which can be simulated. The behavior of the entity is described by the architecture. Multiple architectures are included in a single entity .a single architecture might be structural while another architecture might be a behavioral description of the design.

5.2.3. Package

The main goal of the package is to encapsulate the elements which can be shared (globally) between two or more design units. A package is a popular storage unit that can be used to hold data to be shared between many entities. Data can be shared through packages where the declaration of data inside the package helps the data to be referenced by further entities.

Each package includes two parts: a package body and a package declaration section. The interface of the package is defined by the package declaration and looks like the same method in which the entity defines the model interface. The actual behavior of the package is specified by the package body in the same approach that the architecture statement does for the model.

5.2.4. Process

The basic execution unit of VHDL is the process. The process can be categorized into single and multiple processes in the entire operations that are conducted in a simulation of a VHDL description.

5.3. VHDL MODELLING BASICS

5.3.1. Constants

The constant objects are names given to a particular value of type. Constants provide the capability to have a well documented model and a model that is easy to update. For example, constants are used when a model needs the same value for several cases. The designer can change the value of the constant and compile which will change the whole cases of instances to reflect the new value of the constant.

5.3.2. Signals

Models are formed by the connection of entities together by using signal objects. The communication of dynamic data between entities is implemented by signals. A declaration of the signal is written is as follows:

SIGNAL signal_name: signal_type [:= initial_value];

Signal name(s) is followed by the keyword SIGNAL. A new signal is created by each signal name. A colon separates between the signal name and signal type. Type of signal refers to the type of information on which the signal consists.

The signal can include an initial value specifier through which the value of the signal can be initialized. It is possible to declare the signal in package declarations, architecture declarations and entity declaration sections. Signals declared in the package are referred to as global signals because they may be shared between entities.

5.3.3. VHDL Operators

There are six categories of predefined operators in the language and these operators can be described as follows:

- Additional operators
- Multiplication operators
- Relational operators
- Logical operators
- Shift operators
- Miscellaneous operators

Each operator has increased precedence starting from the category (1) to (5). Operators located in the same classification have the same precedence and the

evaluation is implemented from left to right. Left to right evaluation can be overridden by the use of parentheses.

5.3.4. Concurrent Signal Assignments

Each assignment statement in a typical programming language including C and C++ implements one after the other in identified order. The statement order of the source file determines the order of implementation. No specified ordering of the assignment statements inside VHDL architecture. The implementation order inside VHDL architecture is only specified by events occurring to signal which the assignment statements are sensitive to.

CHAPTER 6

FPGA BASED AM RECEIVER DESIGN AND IMPLEMENTATION

Design and Implementation of FPGA based AM receiver consists of three parts: selection of hardware design components, algorithmic design and simulation of software parts in MATLAB and actual implementation of VHDL source code in the FPGA for actual real world tests.

6.1. HARDWARE COMPONENTS

There are mainly three components in hardware part. MIMAS FPGA board, LM4550 soundcard and IO breakout board which adapts soundcard with PMOD connectors to the FPGA main board. Other than these three hardware components, there are interface cables: one USB-to-serial adapter cable for sending commands to the FPGA board, one USB cable for FPGA programming and two audio extension cables for connecting the analog sound input and outputs of LM4550 soundcard to the PC soundcard. The general view of the system is shown in Figure 6.1.

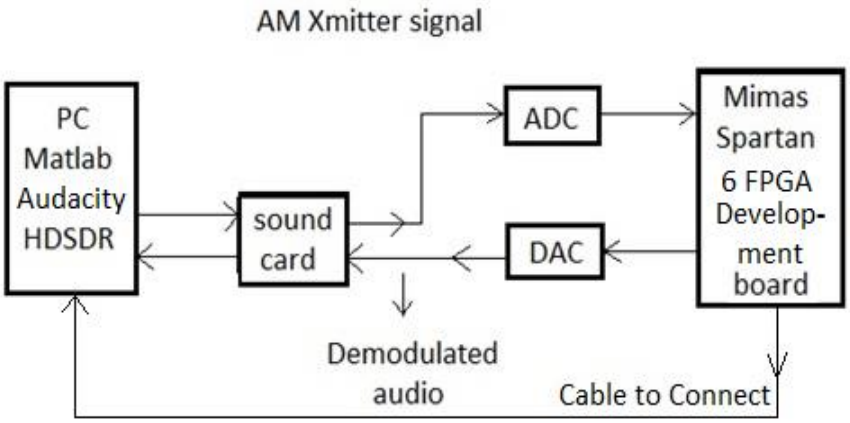


Figure 6.1. General view of FPGA AM receiver.

In Figure 6.1, a modulated signal produced by the Audacity played back wav file which is produced using Matlab codes which will be defined later, is sent through PC's soundcard line-output to the line-in of FPGA's soundcard LM4550. The ADC of the LM4550 digitizes these audio signals in 16-bit, 48KSps, stereo format. The two channels of the stereo audio interface provides a means to represent a complex signal. So, the efective bandwidth of the AM radio signal is $\pm$24KHz as Nyquist criteria for sampling complex signals suggests. Thus the total bandwidth of the processed complex signal will be 48KHz. This complex signal which incorporates two DSB-AM stations where each one plays two 10 second duration of different music on two different frequencies continously, is processed and demodulated by the FPGA fabric which is programmed appropriately for this purpose. So, all the processing of the signal after digitization is carried on by FPGA fabric which can be redefined with software (VHDL in this case). Which of the two stations are selected is determined by a frequency setting command sent through the serial link provided by the virtual serial communication port. The demodulated signal is then sent through the DAC part of the LM4550 soundcard at 16-bits, 48KSps stereo format. Despite the fact that the uplink is a two channel stereo audio stream, the resulting signal is a mono signal in real format and this mono signal is simply repeated in the two channels. However, there is a source select module programmed in the FPGA, which takes commands through built-in switches on the FPGA mainboard and thus the stereo up-link is useful in representing the complex signals that the different part of FPGA AM RX module has. The source select module diverts the inter block signals to output according to the commands from the swtiches and gives the opportunity of seeing the different processing stages of the signal. Also, there is a clipping indicator module in the FPGA fabric, which shows the level (actually an indication of whether clipping occurs or not) of the signal at the input of every signal processing block in the AM receviver module. This indicator is useful in seeing that the level of the signal at the input of that stage is healthy or not. The clipping indicator module uses on-board LEDs for this purpose.

The resulting receiver signals are then monitored, recorded and analyzed at the PC using MATLAB scripts and HDSDR program which is a third party free SDR software used in amateur radio projects.

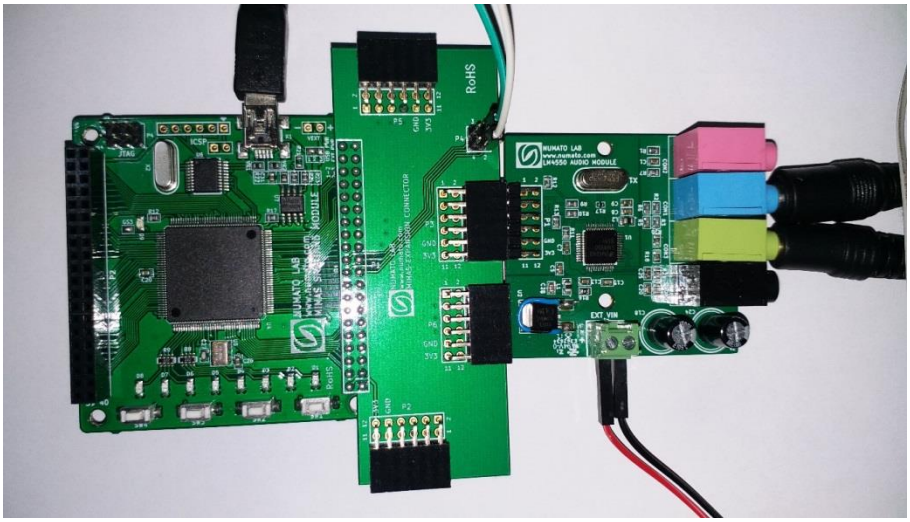


Figure 6.2. Photo of the FPGA AM receiver.

Shown in Figure 6.2 is a photo of the actual working prototype of the system. Here, connecting cables, LM4550 Audio extension card, IO breakout board and main FPGA board is clearly seen. Hardware components of the FPGA AM receiver are given in the next sections in detail.

6.1.1. MIMAS - Spartan 6 FPGA Development Board

6.1.1.1. Introduction

Mimas is easy to use the Development of FPGA board presenting Xilinx Spartan-6 FPGA. Mimas has been designed to learn and experiment design of the system with FPGAs.

This developed board presenting Xilinx XC6SLX9 TQG144 FPGA with a maximum of 70 user IOs.

The USB 2.0 interface offers easy and quick configuration download to the on-board SPI flash. There is no need for a special downloader cable or programmer to download a bitstream to the board [20].

Figure 6.3. Mimas – spartan 6 FPGA development board.

6.1.1.2. Applications

- Product Prototype Development
- Home Networking
- Signal Processing
- Wireless and Wired Communication
- An educational tool for university and school [20].

6.1.1.3. Board Features

- FPGA: Spartan-6 XC6SLX9 in TQG144 package
- Flash memory: 16 Mb SPI flash memory (M25P16)
- 100MHz CMOS oscillator
- USB 2.0 interface for On-board flash programming
- FPGA configuration via JTAG and USB
- 8 LEDs and four switches for user-defined purposes
- 70 IOs for user-defined purposes
- Onboard voltage regulators for single power rail operation [20].

6.1.2. LM4550 Audio Expansion Module

6.1.2.1. Introduction

This Audio module features LM4550, an audio codec for PC systems that is completely compliant and implements the analog concentrated functions of the AC'97 Rev 2.1 architecture. LM4550 uses 18-bit Sigma-Delta ADCs and DACs to create a high-quality stereo audio output [21].

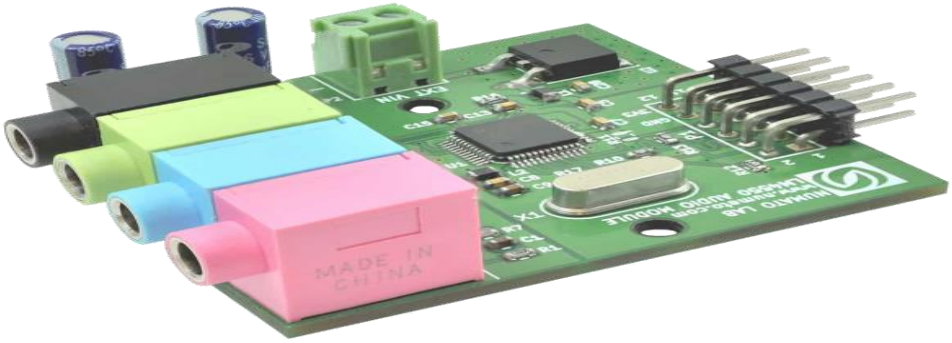


Figure 6.4. LM4550 AC’97 audio expansion module.

6.1.2.2. Applications

- Product Prototype Development
- Audio Record/Playback Systems
- Media players [21].

6.1.2.3. Board Features

- One 2×6 pin Expansion connector
- AC'97 Rev 2.1 Compliant
- 90 dB Dynamic Range
- Stereo Headphone Amp With Separate Gain Control
- Dimension: 50mm x 46mm [21].

6.1.3. IO Breakout Board

6.1.3.1. Introduction

This product is an IO breakout solution for Mimas Sparta6 development board. This product helps Mimas IOs to be categorized into smaller 2x6 headers which may enable easy attachment for the other peripheral expansion modules. It features four 2×6 extension connector [22].

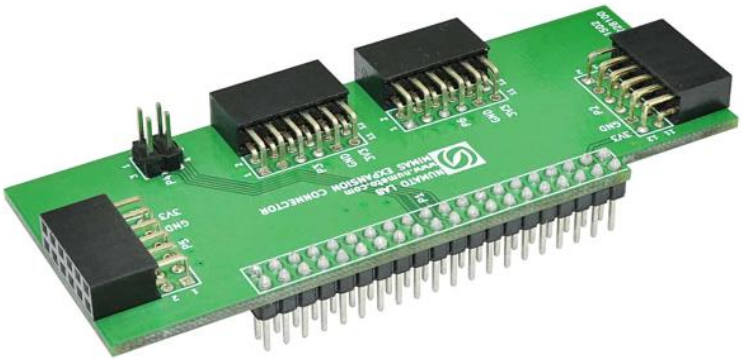


Figure 6.5. IO Breakout module for mimas.

6.1.3.2. Board Features

- Four 2x6 pin expansion connector.
- Can be connected to any side of Mimas.
- Dimension: 34.3mm X 88.1mm [22].

6.2. PROGRAMS

In this study, third party software packages are used in the various stages. These programs and their role in the study are given in the next sections in detail.

6.2.1. Matlab

The significance to use MATLAB in our study is that realistic implementation of SDR must include some equipment, for example, a high-speed A/D converter, a powerful signal processor. This equipment makes the hardware platform very expensive for students who study radio communication. Therefore, we used Matlab in our study and radio signal frequency is limited in the band of audio. One Matlab session has been through the setup of the receiver. Also, Matlab is used to complete all the modulation and demodulation studies. When this system is used, the user needs to only select the modulation and demodulation and corresponding factors.

6.2.2. HDSDR

HDSDR is SDR program which is used to listen into radio, analysis of spectrum and analyse the results. It enjoys by waterfall and varied range separated from each other of the input and output signal. It prevent noise to accomplish the lower speed waterfall spectrum and receive and transmit the signals prepared by similar Matlab scripts. HDSDR monitors and records the waveform produced by PC's sound card. As well as, it works to record and playback RF, IF and AF WAV files with recording scheduler. Therefore, HDSDR software allows a user to enter the mode and global offsets to sync properly the pitch between the radio and SDR audio.

6.2.3. Audacity

It is used as a recording and playback program. It can be used easily as a powerful audio editing and recording package. It allows to record voices and edit recorded voices to correct any mistakes in voices and to combine some sound recordings from many resources including music, interviews, or other recording of sound. Audacity allows to export the recordings in MP3 files format and because of this, it is suitable to produce podcasts.

6.3. MATLAB SIMULATIO CODES

In the first place the test signal used in the experiments, which is played through PC soundcard using Audacity program is generated by the help of a MATLAB script. The code is listed in Appendix B.1. The code is very straightforward and comments in the code explains itself. The MATLAB program firstly takes two audio sample files in wave format which are prepared as 10 second duration mono music sampled at 8KSps format wave files. It up-samples and interpolates to 48KSps each and uses in the modulation of two different AM stations whose frequency is determined by the parameters in the code and so can be changeable. The resulting modulated waveform is complex thus it is recorded in stereo format. This sample signal is then used in both actual operational testing and MATLAB simulation of the system.

Another script is used to design filters used in the system and derived coefficients are then transferred to the FIR filter IP component through a coefficient file. The MATLAB script used for this purpose is listed in Appendix B.2. The program shows the frequency response of the designed filter as a graphic and stores the coefficients of the filter in a file which will be used in the FIR filter IP of the FPGA AM receiver. Here, number of the coefficients, sampling rate and cut-off frequency of the filter can be changed as desired.

The design of the algorithms underlying the principles of FPGA AM receiver and simulation of the system is achieved using a MATLAB script. Then, the algorithms are transferred to FPGA after recoded in VHDL. The VHDL codes of the FPGA AM receiver will be discussed later. The listing of the simulation code is given in Appendix B.3. This code is also very straight forward and explained by comment lines well. It takes test signal in wave file format, demodulates one of the AM stations whose frequency is set by a parameter in the code and using squaring method (envelope detection) demodulates, filters and records the resulting signal as a 48KSps mono wave file so that it can be played back and listened later. Also, products obtained at the various stages of demodulation process is recorded in separate wave files for seeing the evolution of the signal and further analysis purposes.

Lastly a MATLAB script listed in Appendix B.4 is used to compare the original modulating waveform with the demodulation result of the either simulation program or FPGA AM receiver which is recorded by HDSDR and post processed with Audacity to cut and synchronize with the original. The waveforms that will be compared is thus recorded in one stereo wave file where left channel carries the demodulated signal and right channel carries original modulating signal. The results will be discussed on the chapter about Results and Discussion.

6.4. VHDL CODE AND BLOCK SCHEMA OF THE SYSTEM

All the VHDL code of the system is given in Appendix C. As with the MATLAB codes, comments explain everything. The block schema given in Figure 6.6 summarize the functioning of the FPGA AM RX and show the relations between different modules. Top module cnt is used as a wire loom for other sub-modules and provides the interconnections between different modules of the system. Also, top module contains necessary codes to arrange clocks and resets used by sub-modules of the system.

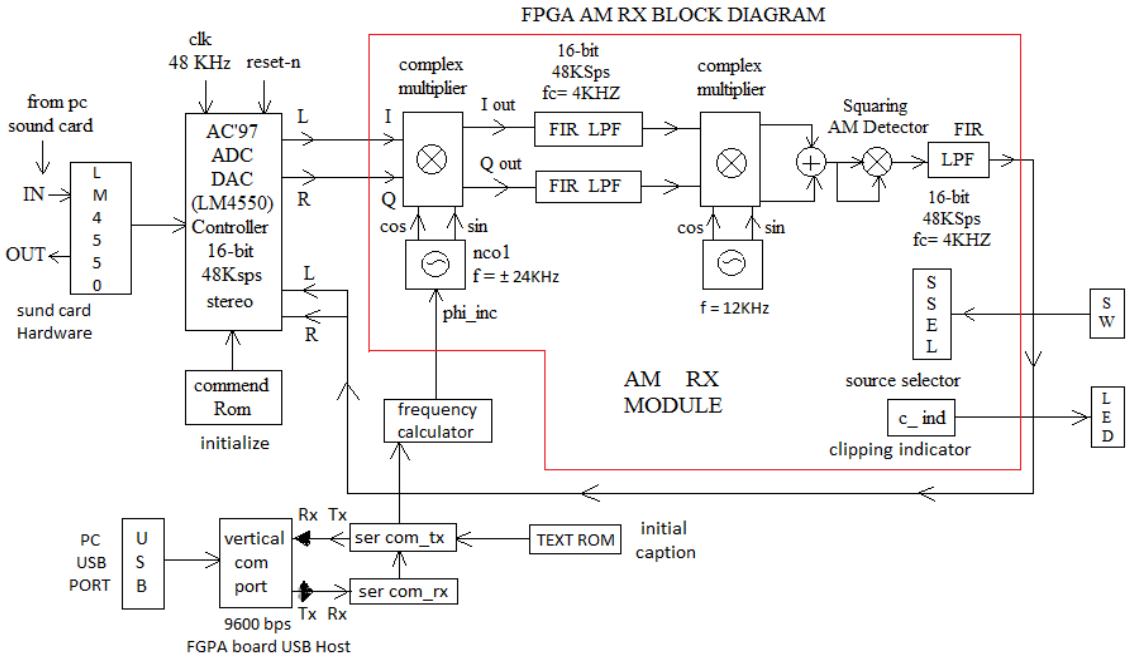


Figure 6.6. Block schema of the FPGA AM RX system.

All the basic AM receiver functions is contained in the AM_RX module. As seen on Figure 6.6, AM_RX module can handle complex signals. Despite the output is also two channels, it is not in complex form. The resulting demodulated signal can only be in real format thus it is copied on both channels. The outputs of different sub-modules in AM_RX module can be directed to the output and since many of them provides complex output when they are selected, the output of the AM_RX module is operated in complex mode. The selection is achieved by built-in switches of the Mimas FPGA board. Also, every sub-module in the signal chain of AM_RX module has its own clipping indicators and outputs of these indicators are directed to built-in LEDs on the board.

Complex input to AM_RX module is firstly frequency shifted by an amount controlled by the first NCO, whose frequency control input (phase increment input) is driven by the frecalc module which in turn takes commands from sercomrx module which provides frequency setting commands received from PC through a USB-to-serial cable using a suitable terminal program such as Termite or PuTTY run on the host PC. All serial data communication is handled by the sercomrx and sercomtx modules in the FPGA design. The received commands by sercomrx module are echoed through sercomtx module to host PC. The command format is f<sign><frequency> where f represents that it is a frequency changing command, sign is either + or – and frequency is a 5-digit integer number in the range 00000 – 23999 which is the absolute value of frequency in Hz. So, the frequency can only be changed in 1Hz increments (frequency resolution is 1Hz).

The complex frequency down shifting operation sets the center frequency of the received station to zero. The complex radio signal then low-pass filtered whose cut-off frequency is set to 4KHz, which is compatible with the bandwidth of the baseband modulating test signal used in the experiments. Then the signal is complex up-shifted to 12KHz using a second fixed frequency NCO and a subsequent complex multiplier IP. This last fixed upshifting operation is necessary for demodulation.

All the FIR filters, NCOs and complex and real multipliers are implemented using ready-made Xilinx IPs which shortens design time extremely and provides high

performance. Every IP used is configured and setup using respective wizards whose use is very comprehensive but explained clearly in respective datasheets of every IP. So, they will not be covered here in detail.

After second frequency shift, the signal being bandlimited to ±4KHz at the center frequency of 12KHz is realized by an adder and then demodulated using a squaring method which uses just a single multiplier. The resulting signal must be low-pass filtered to get rid of the high frequency products generated in the non-linear squaring process. After this the original modulating signal is obtained clearly.

All three FIR filters in the design are identical in input and output sampling rates (48KSps), bit resolution (16-bits), number of coefficients (255), coefficient resolutions (16-bits) and cut-off frequency (Fc=4KHz). So, after their coefficients found using a MATLAB script, they are implemented using the FIR filter IP design wizard and the resulting module is instantiated (copied) three times.

Signal digitization and reproduction is carried over by the LM4550 based soundcard. So, signal exchange between PC and FPGA AM RX is through audio cables in analog format. For the management of the LM4550 soundcard, a VHDL module is written by the help of the respective datasheet. The management module resets, sets up and configures the LM4550 chip prior usage and then handles the data streaming between LM4550 and Spartan6 FPGA.

As said before frequency control is achieved through a USB-to-TTL serial cable and using a terminal program in the PC. The serial link cable also provides the auxiliary 5V supply that the LM4550 soundcard necessitates since Mimas FPGA board only supplies 3.3V.

According to the design utilization summary report generated in the synthesis process by Xilinx ISE webpack suite, out of 1430 slices 702 is used which corresponds to 49% utilization. Out of 32 RAMB16WER ram blocks 30 is used which corresponds to 93% utilization. And lastly, out of 16 DSP48A1 DSP blocks 15 is used which corresponds to 93% utilization. So, in the light of these information the

design is said to be RAM and DSP intensive. It is normal for a SDR design implemented in FPGA because their signal chain usually requires RAM and DSP intensive filter and multiplier operations.

6.5. RTL DIAGRAMS OF THE SYSTEM

RTL diagrams are useful in showing the internals of a design in FPGA. They provide an easy and quick way of understanding the operation of the design. They are also used as an alternative tool in the debugging process of the design. Most important modules of the system are presented here to show their operation. These are top module cnt and am_rx modules.

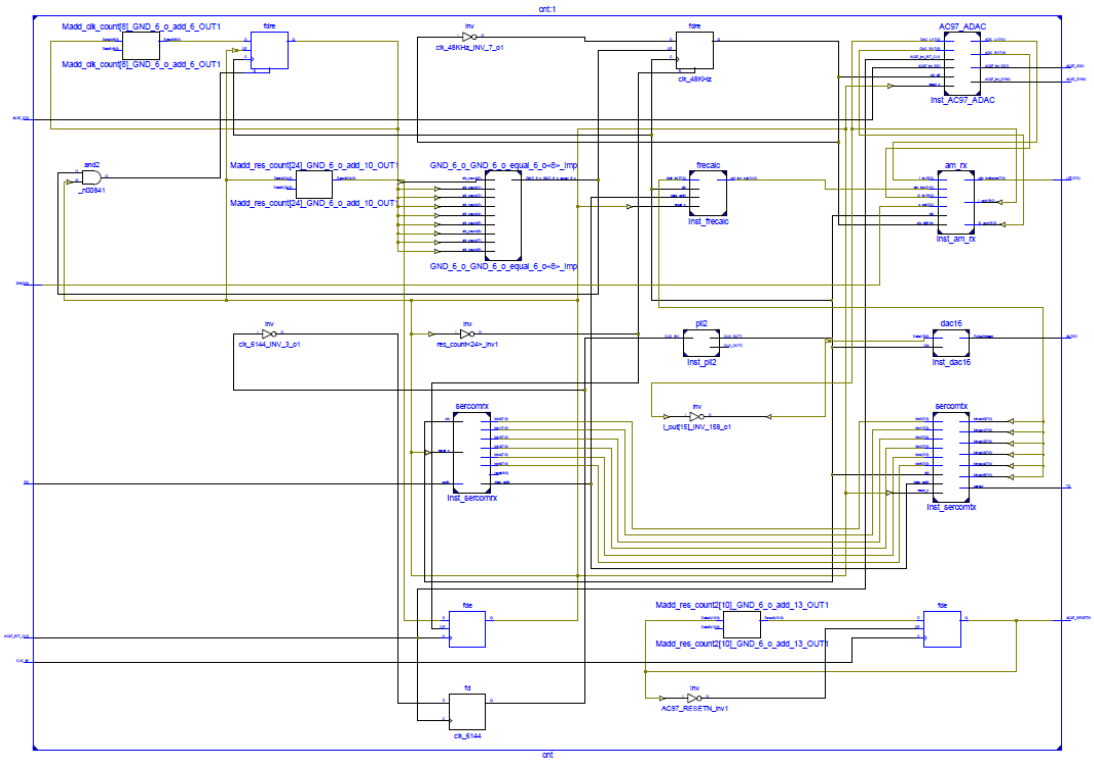


Figure 6.7. RTL schematic of top module cnt.

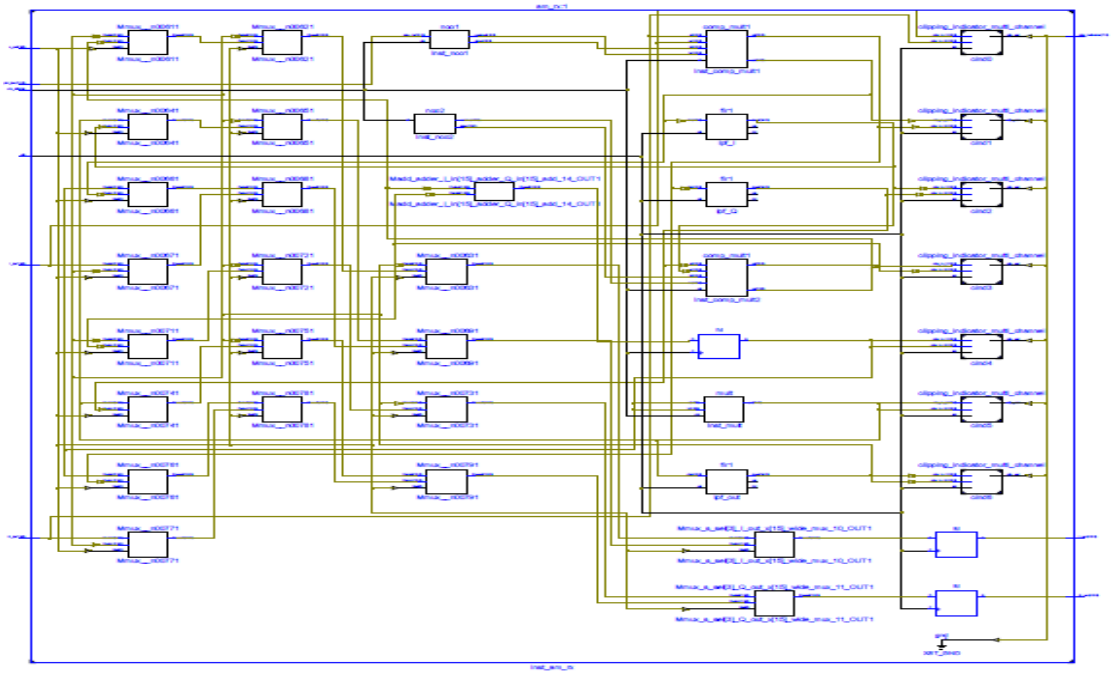


Figure 6.8. RTL schematic of am_rx module.

CHAPTER 7

RESULTS AND DISCUSSION

The system is simulated and tested using two 10s sound recordings firstly recorded at 8KSps and then upsampled to 48KSps to cope with the modulation process. Each recording incorporates music which fill the 4KHz spectra in a normal distrubution so that test signals provide a similar result with a white noise source. The recording is used in the modulation of two AM stations put in different frequencies with the test signal A1 on fc=5KHz and A2 on fc=-15KHz in the complex signal at 48KSps. Modulation is done using a MATLAB script whose listing given on Appendix B1. This modulated complex test signal then used in the simulation of demodulation using the code listed in Appendix B3. The test signal is also used in actual real world experiments carried on FPGA AM RX system. This is achieved by playing the test signal in a continuous loop outputted to the soundcard of PC using Audacity as the player. After demodulation by the FPGA AM RX, the resulting waveform is captured by the soundcard of the PC and monitored and recorded by the HDSDR SDR program. The resulting waveforms from simulation and test are post-processed before put into analysis using a MATLAB script which is listed on Appendix B4. Post-processing incorporates normalizing and syncronization of the resulting waveform to respective input test signal, either A1 or A2. Post-processing is carried on using Audacity. The post-process result is then recorded in 2-channel stereo format in order to preserve the synchrocity. In this format the top signal (Left channel) holds the demodulated waveform and the bottom signal (Right channel) holds the original test signal. The analysis operation provides three results: wav recording of difference signal between demodulation and original test signal, rms level of error signal (obtained from difference waveform using a 10s window), Signal-to-Noise Ratio (SNR) in dB calculated from the rms error and rms level of the original test signal. Demodulation waveform, original test signal waveform and difference waveform combined in a single graphic time plot for each test signals and

for each of the simulation and test are presented in Figures 7.1-4. Also the analysis results for each of the two test waveforms for each of the simulation and test is listed in Table 7.1 for comparison.

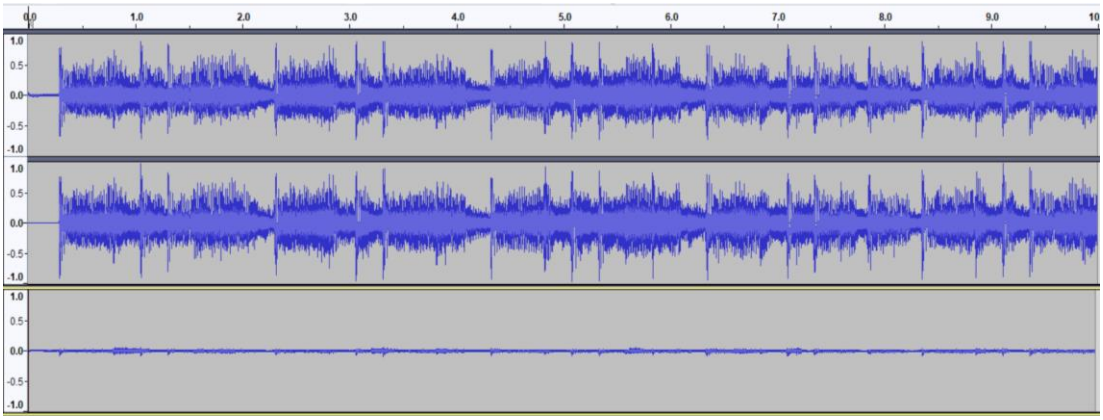


Figure 7.1. FPGA AM RX system test results with A1 test signal: top signal is output demodulated waveform, middle test signal A1 at the input and the bottom signal is difference between two.

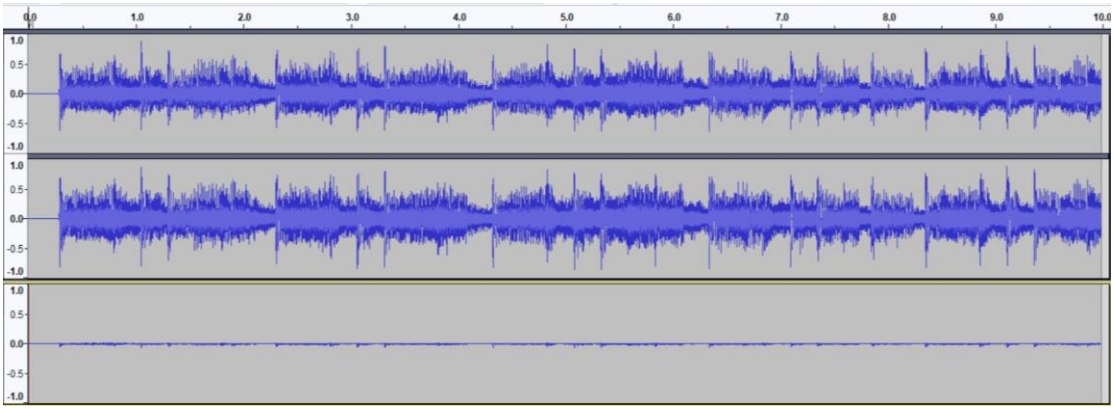


Figure 7.2. Matlab simulation results with A1 test signal: top signal is output demodulated waveform, middle test signal A1 at the input and the bottom signal is difference between two.

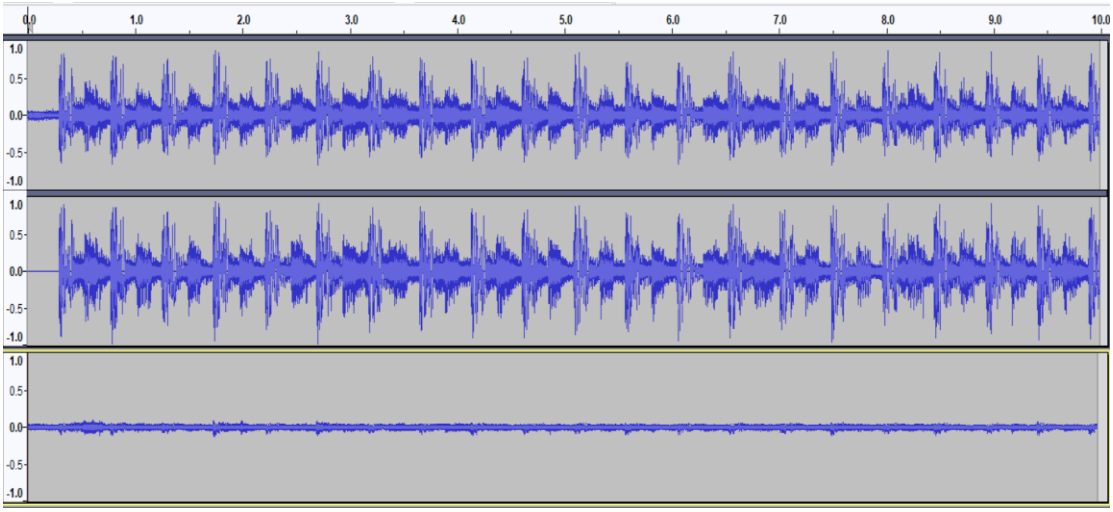


Figure 7.3. FPGA AM RX system test results with A2 test signal: top signal is output demodulated waveform, middle test signal A2 at the input and the bottom signal is difference between two.

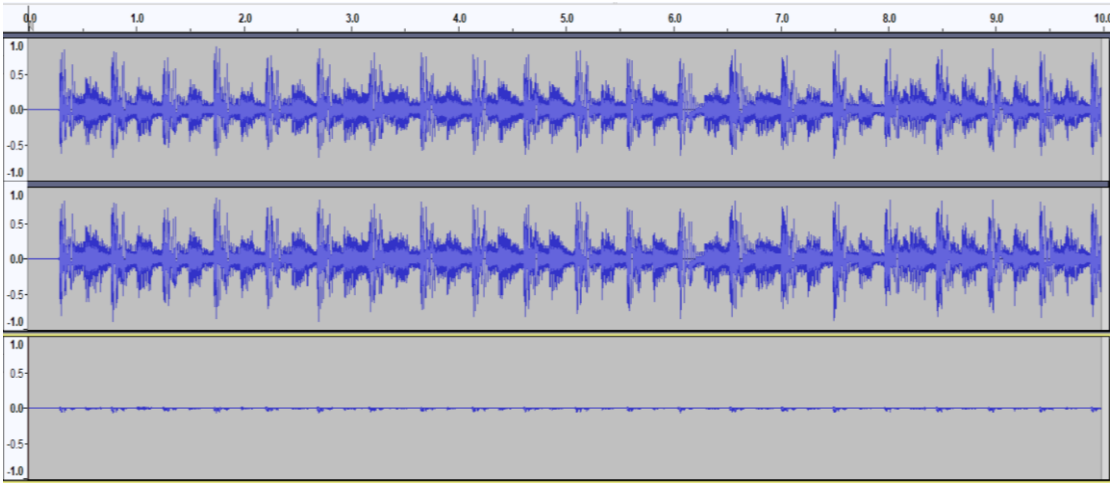


Figure 7.4. Matlab simulation results with A2 test signal: top signal is output demodulated waveform, middle test signal A2 at the input and the bottom signal is difference between two.

Table 7.1. Test results.

Test Signal	Error (rms)	SNR (dB)
A1_test	9.235118e-03	26.24
A1_sim	4.700772e-03	31.14
A2_test	16.90000e-03	20.87
A2_sim	6.003603e-03	28.98

As it is seen from figures 7.1 and Table 7.1, simulation results are slightly better than actual real world tests of FPGA AM RX system. This is normal because, actual real world tests incorporate more noise coming from different sources namely: electrical noise on the audio cables, inherent noise of ADCs and DACs of soundcards both PC and LM4550 soundcards and samplerate differences between receiving and transmitting pairs of ADCs and DACs. Also, there is noise coming from the inaccurate normalization and synchronization processes done in the post-processing of the results before analysis. As SNRs from each test run is compared, higher SNR means better. Best performance is achieved in simulations done with the test signal A1. Simulation with test signal A2 provided a lower performance compared to simulation with test signal A2. This is also true for the actual real world tests. This is attributed to the relatively low modulation depth used in the test signal A2. Usually SNR decreases with decreasing modulation depth so this is normal. When actual real world values are compared to simulations for each of the test signal it is seen that real world SNR results are slightly lower than simulation SNRs. The difference between SNRs for test signal A1 is 4.9dB and for test signal A2 it is 8.11dB. Again tests with A1 is better than tests with A2.

Lastly, test results provide a SNR higher than 20dB, which is an acceptable level for an AM receiver. So, the designed and implemented FPGA AM recevier can be assumed successful in demodulation of AM signals.

CHAPTER 8

CONCLUSION

In this study, an AM receiver using SDR techniques is designed and implemented in FPGA. The main purpose of the study is to provide a cheap and simple FPGA based platform for teaching and learning of SDR basics. Test and simulation results proved on the signals A1 and A2 that higher SNR means better. And through a comparison between them, noted that A2 real world SNR results are slightly lower than A1 simulation SNRs.

Where, these results proved that tests with A1 is better than tests with A2. This is attributed to the relatively low modulation depth used in the test signal A2 and too this is normal because, actual real world tests incorporate more noise coming from different sources namely: electrical noise on the audio cables, inherent noise of ADCs and DACs of soundcards both PC and LM4550 soundcards and sample rate differences between receiving and transmitting pairs of ADCs and DACs.. Also, test and simulation results prove FPGA AM RX system a useful candidate for AM demodulation and reception. The designed and implemented FPGA AM RX system is also a good utility in the education of basic SDR principles which is the focus of this study.

So, the designed and implemented FPGA AM receiver can be assumed successful in demodulation of AM signals. It can be used in teaching the radio signal processing techniques using FPGAs. As the system is also suitable to be used with any soundcard based SDR frontend such as Softrock Ensemble receiver for HF and SW bands. It can be used in a standalone fashion if a microcontroller is used to send commands or a user interface for this purpose may be designed in the FPGA. The design costumes most of the RAM and DSP blocks but has unused logic slices sufficient for enhancements like that.

As a future work, other modulation types and methods can be added to this system to show the principles behind them. A frontend and a user interface may be designed for it to make it a more practical SDR system which can be used for amateur and research purposes as well as in the education of communication engineering students.

REFERENCES

1. T. S. Rappaport, "Wireless Communication - Principle and practice, 2nd ed.", ***Prentice Hall***, (2002).

2. Dierker, M., "Chapter Three: Amplitude Modulation Fundamentals", ***Press***, 93 117 (2007).

3. Lee, Thomas H., "The Design of CMOS Radio-Frequency Integrated Circuits, 2nd ed.", ***Cambridge University,*** UK, (2004).

4. Lewis, C., "Wireless Radio: A History, 2nd ed.", ***McFarland & Company,*** US, (2006).

5. Harnani, H., Cik K. H. Y., and Nor, I. S. B., "Low Complexity SDR Transceiver Design using Simulink, Matlab and Xilinx", ***IEEE International Conference on ICT Convergence (ICTC),*** (2012).

6. Parikh, K. S., Shahana, K., and Gupta, R. K., "SDR - Implementation of Low Frequency Trans-Receiver on FPGA", ***IEEE International Conference on Signal Processing and Integrated Networks (SPIN),*** (2014).

7. Jiang-Tao, G., Chuan-Wu, T., "An Algorithm of Software Defined Radio Channel Processing Based on FPGA", ***IEEE International Conference on Wireless Communication and Sensor Network,*** (2014).

8. Kohno, R., "Prespective of Software Radoi:Spatial and Temporal Communication Theory Using Adaptive Array Antena for Mobile Radio Communications", ***Microwave Workshops and Transsion(MWE'97),*** *25-31*, PacificoYokohama, Dec (1997).

9. Mannan, P. M., "Framework for the design and implementation of software define radio on wireless communication system", Master Thesis, ***University of Akron***, December (2005).

10. Lackey, R. J. and Upmal, D. W., "Speakeasy: The Military Software Radio", ***IEEE*** (2016).

11. Carlson, A. B., "Communications Systems An Introduction to signals and Noise, 4th ed.", ***McGraw-Hill Higher Education***, (2002).

12. Gibson, J. D., "The Communications Handbook, 2nd ed.", ***CRC Press***, (2002).

13. Hosking, R. H., "Software Defined Radio Handbook (Notes Gathering), 8th ed.", ***Press*** (2010).

14. Anderson, J. B., and Rolf, J., “Understanding Information Transmission”, ***Wiley-IEEE Press,*** (2005).

15. Giannini, V., Craninckx, J. and Baschirotto, A., “Baseband Analog Circuits for Software Defined Radio”, ***Springer,*** (2008).

16. Mehta, V. K. and Mehta, R., “Chapter 16: Modulation and Demodulation, Principles of Electronics”, ***S. Chand & Company***, Ram Nagar, New Delhi (2014).

17. Dierker, M., “Chapter Three: Amplitude Modulation Fundamentals”, ***Press***, 93-117 (2007).

18. Internet: Wikipedia, “About Register-transfer_leve”, **https://en.wikipedia.org/wiki/Register-transfer_level** (2020).

19. Dominik, M., Katarína, J., “VHDL structural model visualization”, ***IEEE EUROCON - International Conference on Computer as a Tool,*** (2011).

20. Internet: Numato, “Mimas - Spartan 6 FPGA Development Board”, **https://numato.com/docs/mimas-spartan-6-fpga-development-board/** (2018).

21. Internet: Numato, “LM4550 Audio Codec Expansion Module”, **https://numato.com/product/lm4550-ac97-stereo-audio-codec-expansion-module/** (2020).

22. Internet: Numato, “IO Breakout Module For Mimas”, **https://numato.com/product/io-breakout-module-for-mimas/** (2020).

APPENDIX A.

DATASHEETS OF ELECTRONIC COMPONENTS (Spartan 6, LM4550)

A1. Datasheet Spartan 6 (xc6slx9-3tqg144)

XILINX®

Spartan-6 Family Overview

DS160 (v2.0) October 25, 2011 **Product Specification**

General Description

The Spartan®-6 family provides leading system integration capabilities with the lowest total cost for high-volume applications. The thirteen-member family delivers expanded densities ranging from 3,840 to 147,443 logic cells, with half the power consumption of previous Spartan families, and faster, more comprehensive connectivity. Built on a mature 45 nm low-power copper process technology that delivers the optimal balance of cost, power, and performance, the Spartan-6 family offers a new, more efficient, dual-register 6-input look-up table (LUT) logic and a rich selection of built-in system-level blocks. These include 18 Kb (2 x 9 Kb) block RAMs, second generation DSP48A1 slices, SDRAM memory controllers, enhanced mixed-mode clock management blocks, SelectIO™ technology, power-optimized high-speed serial transceiver blocks, PCI Express® compatible Endpoint blocks, advanced system-level power management modes, auto-detect configuration options, and enhanced IP security with AES and Device DNA protection. These features provide a low-cost programmable alternative to custom ASIC products with unprecedented ease of use. Spartan-6 FPGAs offer the best solution for high-volume logic designs, consumer-oriented DSP designs, and cost-sensitive embedded applications. Spartan-6 FPGAs are the programmable silicon foundation for Targeted Design Platforms that deliver integrated software and hardware components that enable designers to focus on innovation as soon as their development cycle begins.

Summary of Spartan-6 FPGA Features

- Spartan-6 Family:
  - Spartan-6 LX FPGA: Logic optimized
  - Spartan-6 LXT FPGA: High-speed serial connectivity
- Designed for low cost
  - Multiple efficient integrated blocks
  - Optimized selection of I/O standards
  - Staggered pads
  - High-volume plastic wire-bonded packages
- Low static and dynamic power
  - 45 nm process optimized for cost and low power
  - Hibernate power-down mode for zero power
  - Suspend mode maintains state and configuration with multi-pin wake-up, control enhancement
  - Lower-power 1.0V core voltage (LX FPGAs, -1L only)
  - High performance 1.2V core voltage (LX and LXT FPGAs, -2, -3, and -3N speed grades)
- Multi-voltage, multi-standard SelectIO™ interface banks
  - Up to 1,080 Mb/s data transfer rate per differential I/O
  - Selectable output drive, up to 24 mA per pin
  - 3.3V to 1.2V I/O standards and protocols
  - Low-cost HSTL and SSTL memory interfaces
  - Hot swap compliance
  - Adjustable I/O slew rates to improve signal integrity
- High-speed GTP serial transceivers in the LXT FPGAs
  - Up to 3.2 Gb/s
  - High-speed interfaces including: Serial ATA, Aurora, 1G Ethernet, PCI Express, OBSAI, CPRI, EPON, GPON, DisplayPort, and XAUI
- Integrated Endpoint block for PCI Express designs (LXT)
- Low-cost PCI® technology support compatible with the 33 MHz, 32- and 64-bit specification.
- Efficient DSP48A1 slices
  - High-performance arithmetic and signal processing
  - Fast 18 x 18 multiplier and 48-bit accumulator
  - Pipelining and cascading capability
  - Pre-adder to assist filter applications
- Integrated Memory Controller blocks
  - DDR, DDR2, DDR3, and LPDDR support
  - Data rates up to 800 Mb/s (12.8 Gb/s peak bandwidth)
  - Multi-port bus structure with independent FIFO to reduce design timing issues
- Abundant logic resources with increased logic capacity
  - Optional shift register or distributed RAM support
  - Efficient 6-input LUTs improve performance and minimize power
  - LUT with dual flip-flops for pipeline centric applications
- Block RAM with a wide range of granularity
  - Fast block RAM with byte write enable
  - 18 Kb blocks that can be optionally programmed as two independent 9 Kb block RAMs
- Clock Management Tile (CMT) for enhanced performance
  - Low noise, flexible clocking
  - Digital Clock Managers (DCMs) eliminate clock skew and duty cycle distortion
  - Phase-Locked Loops (PLLs) for low-jitter clocking
  - Frequency synthesis with simultaneous multiplication, division, and phase shifting
  - Sixteen low-skew global clock networks
- Simplified configuration, supports low-cost standards
  - 2-pin auto-detect configuration
  - Broad third-party SPI (up to x4) and NOR flash support
  - Feature rich Xilinx Platform Flash with JTAG
  - MultiBoot support for remote upgrade with multiple bitstreams, using watchdog protection
- Enhanced security for design protection
  - Unique Device DNA identifier for design authentication
  - AES bitstream encryption in the larger devices
- Faster embedded processing with enhanced, low cost, MicroBlaze™ soft processor
- Industry-leading IP and reference designs

Spartan-6 FPGA Feature Summary

Table 1: **Spartan-6 FPGA Feature Summary by Device**

Device	Logic Cells[1]	Configurable Logic Blocks (CLBs)			DSP48A1 Slices[3]	Block RAM Blocks		CMTs[5]	Memory Controller Blocks (Max)[6]	Endpoint Blocks for PCI Express	Maximum GTP Transceivers	Total I/O Banks	Max User I/O
		Slices[2]	Flip-Flops	Max Distributed RAM (Kb)		18 Kb[4]	Max (Kb)						
XC6SLX4	3,840	600	4,800	75	8	12	216	2	0	0	0	4	132
XC6SLX9	9,152	1,430	11,440	90	16	32	576	2	2	0	0	4	200
XC6SLX16	14,579	2,278	18,224	136	32	32	576	2	2	0	0	4	232
XC6SLX25	24,051	3,758	30,064	229	38	52	936	2	2	0	0	4	266
XC6SLX45	43,661	6,822	54,576	401	58	116	2,088	4	2	0	0	4	358
XC6SLX75	74,637	11,662	93,296	692	132	172	3,096	6	4	0	0	6	408
XC6SLX100	101,261	15,822	126,576	976	180	268	4,824	6	4	0	0	6	480
XC6SLX150	147,443	23,038	184,304	1,355	180	268	4,824	6	4	0	0	6	576
XC6SLX25T	24,051	3,758	30,064	229	38	52	936	2	2	1	2	4	250
XC6SLX45T	43,661	6,822	54,576	401	58	116	2,088	4	2	1	4	4	296
XC6SLX75T	74,637	11,662	93,296	692	132	172	3,096	6	4	1	8	6	348
XC6SLX100T	101,261	15,822	126,576	976	180	268	4,824	6	4	1	8	6	498
XC6SLX150T	147,443	23,038	184,304	1,355	180	268	4,824	6	4	1	8	6	540

Notes:

1. Spartan-6 FPGA logic cell ratings reflect the increased logic cell capability offered by the new 6-input LUT architecture.
2. Each Spartan-6 FPGA slice contains four LUTs and eight flip-flops.
3. Each DSP48A1 slice contains an 18 x 18 multiplier, an adder, and an accumulator.
4. Block RAMs are fundamentally 18 Kb in size. Each block can also be used as two independent 9 Kb blocks.
5. Each CMT contains two DCMs and one PLL.
6. Memory Controller Blocks are not supported in the -3N speed grade.

A2. Datasheet LM4550

LM4550B AC '97 Rev 2.1 Multi-Channel Audio Codec With Stereo Headphone Amplifier, Sample Rate Conversion and TI 3D Sound

1 Features

- AC '97 Rev 2.1 Compliant
- High Quality Sample Rate Conversion From 4 kHz to 48 kHz in 1 Hz Increments
- Supports up to 6 DAC Channel Systems With Multiple LM4550Bs or With Other TI LM45xx Codecs
- Unique TI Chaining Function Shares a Single Controller SDATA_IN Pin Among Multiple Codecs
- Stereo Headphone Amp With Separate Gain Control
- TI's 3D Sound Stereo Enhancement Circuitry
- Advanced Power Management Support
- External Amplifier Power-Down (EAPD) Control
- PC Beep Passthrough to Line Out During Initialization or Cold Reset
- Digital 3.3-V and 5-V Supply Options
- Extended Temperature: $-40°C \le T_A \le 85°C$
- Key specifications
  - Analog Mixer Dynamic Range, 97 dB (Typical)
  - DAC Dynamic Range, 89 dB (Typical)
  - ADC Dynamic Range, 90 dB (Typical)
  - Headphone Amp THD+N at 50 mW, 0.02% (Typical) into 32Ω

2 Applications

- Desktop PC Audio Systems on PCI Cards, AMR Cards, or With Motherboard Chips Sets Featuring AC Link
- Portable PC Systems as on MDC Cards, or with a Chipset or Accelerator Featuring AC Link
- General Audio Frequency Systems Requiring 2, 4 or 6 DAC Channels and/or up to 8 ADC Channels
- Automotive Telematics

3 Description

The LM4550B device is an audio codec for PC systems which is fully PC99 compliant and performs the analog intensive functions of the AC '97 Rev 2.1 architecture. Using 18-bit Sigma-Delta ADCs and DACs, the LM4550B provides 90 dB of Dynamic Range.

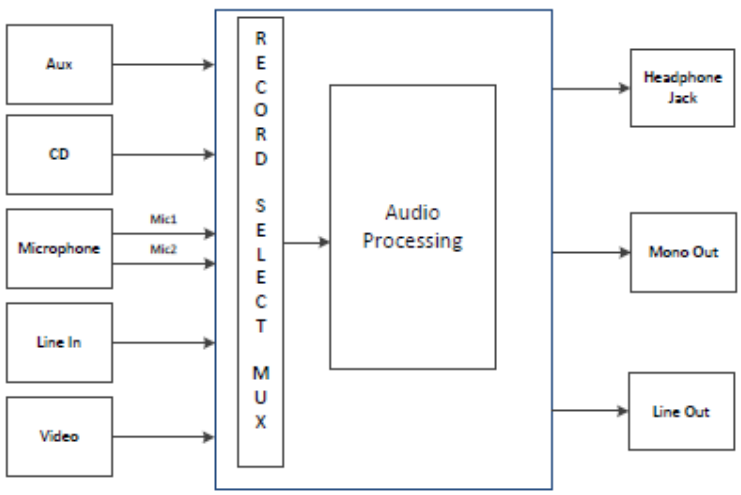
The LM4550B was designed specifically to provide a high quality audio path and provide all analog functionality in a PC audio system. It features full duplex stereo ADCs and DACs and analog mixers with access to 4 stereo and 4 mono inputs.

Device Information(1)

PART NUMBER	PACKAGE	BODY SIZE (NOM)
LM4550B	LQFP (48)	7.00 mm × 7.00 mm

(1) For all available packages, see the orderable addendum at the end of the data sheet.

Simplified Block Diagram



5 Description (continued)

Each mixer input has separate gain, attenuation and mute control and the mixers drive 1 mono and 2 stereo outputs, each with attenuation and mute control. The LM4550B provides a stereo headphone amplifier as one of its stereo outputs and also supports TI's 3D sound stereo enhancement and a comprehensive sample rate conversion capability. The sample rate for the ADCs and DACs can be programmed separately with a resolution of 1 Hz to convert any rate from 4 kHz to 48 kHz. Sample timing from the ADCs and sample request timing for the DACs are completely deterministic to ease task scheduling and application software development. These features together with an extended temperature range also make the LM4550B suitable for non-PC codec applications.

The LM4550B features the ability to connect several codecs together in a system to provide up to 6 simultaneous channels of streaming data on output frames (controller to codec) for surround sound applications. Such systems can also support up to 8 simultaneous channels of streaming data on input frames (codec to controller). Multiple codec systems can be built either using the standard AC Link configuration (that is, of one serial data signal to the controller per codec) or using a unique TI feature for chaining codecs together. This chain feature shares only a single data signal to the controller among multiple codecs.

The AC '97 architecture separates the analog and digital functions of the PC audio system allowing both for system design flexibility and increased performance.

APPENDIX B.

MATLAB CODE LISTINGS

B1. Modulation (Test Signal Generation)

```
% (DSB-WC) Mod. with MUSIC by B. ERKAL 2020
% AM transmitter code by Bilgehan ERKAL
% Karabuk 2020
clear all;

% sound file 1 loading (4Khz mono (8KSps))
[iff1 , afs]=audioread('a1.wav');
[y1,~]=size(iff1);
% upsample x6 (8x6=48Khz)
yu1=upsample(iff1,6);
% Baseband signal is filtered and normalized
yu1=filter(fir1(128,4e3/24e3),1,yu1);
yu1=yu1./(1.01*max(abs(yu1)));
audiowrite('a1_48k.wav', yu1, 48e3);

% sound file 2 loading (4Khz mono (8KSps))
[iff2 , afs]=audioread('a2.wav');
[y1,~]=size(iff1);
% upsample x6 (8x6=48Khz)
yu2=upsample(iff2,6);
% Baseband signal is filtered and normalized
yu2=filter(fir1(128,4e3/24e3),1,yu2);
yu2=yu2./(1.01*max(abs(yu2)));
audiowrite('a2_48k.wav', yu2, 48e3);

fs=48e+3;       % sampling frequency
ts=1/fs;        % sampling interval
t=0:ts:10-ts;   % time axis

% carrier parameters: amplitude, frequency and phase
C1=1; C2=1;
fct1=5e+3;
fct2=-15e+3;
tetac1=0*(pi/180);
tetac2=0*(pi/180);

% carrier signal
ct1=C1*exp(2*1i*pi*fct1*t+tetac1);
ct2=C2*exp(2*1i*pi*fct2*t+tetac2);

% Complex AM (DSB-WC) signal
m=0.2*(yu1'+3).*ct1+0.2*(yu2'+3).*ct2;

% IF signal is recorded in wav file
% IF normalized
m=m./(1.1*max(abs(m)));
```

```
audiowrite('DSB_WC.wav', [real(m)', imag(m)'], fs);
```

B2. Filter Design

```
clear all;
% filter cut at 4KHz 48KSps (24KHz)
no_coeff = 254; % number of taps (coefficients)
fc = 4e3;            % cut-off frequency
nbw = 24e3;        % nyqusit bandwidth (limit)
type ='low';         % type of the filter
% FIR filter structure
yu=fir1(no_coeff,fc/nbw,type);
% Filter coefficients are cast in 16-bit signed integers for using in FIR IP in
FPGA
z=int16(32767*(yu./max(abs(yu))));
freqz(yu);     % Bode-plot of frequency response
% coefficients of the designed filter are stored in a file
fid = fopen('exp.txt','w');
fprintf(fid,'%i ',z);
fclose(fid);
```

B3. Simulation

```
% (DSB-WC) Demod. with MUSIC by B. ERKAL 2020
% AM receiver code by Bilgehan ERKAL
% Karabuk 2020
clear all;

% complex IF file loading (48Khz stereo)
[iff1 , afs]=audioread('DSB_WC.wav');
[y1,~]=size(iff1);
% complex conversion
yu1=iff1(1:y1,1)'+1i*iff1(1:y1,2)';

% IF signal is normalized
yu1=yu1./(1.01*max(abs(yu1)));

fs=afs;         % sampling frequency
ts=1/fs;        % sampling interval
t=0:ts:10-ts;   % time axis
% carrier parameters: amplitude, frequency and phase
C1=0.1;
fct1=-5e+3;
tetac1=0*(pi/180);

% carrier signal
```

```
ct1=C1*exp(2*1i*pi*fct1*t+tetac1);
ct2=C1*exp(2*1i*pi*12e3*t+tetac1);

% demodulation
% complex frequency downshift operation
iffc=yu1.*ct1;

% zero IF cut at 4KHz
yu=filter(fir1(255,4e3/(24e3)),1,iffc);
% complex result is normalized and recorded
yu=yu./(1.01*max(abs(yu)));
audiowrite('res1.wav', [real(yu)',imag(yu)'], fs);

% AM detection
% complex upshifting for detector IF ofset
yu=yu.*ct2;
% complex result is normalized and recorded
yu=yu./(1.01*max(abs(yu)));
audiowrite('res2.wav', [real(yu)',imag(yu)'], fs);

% AM demodulation using squaring method
% first complex IF is realized
dem=real(yu)+imag(yu);
% real IF is normalized and recorded
dem=dem./(1.01*max(abs(dem)));
audiowrite('res3.wav', dem, fs);
% actual demodulation of real IF signal is accomplished here
dem=dem.*dem;
% Raw demodulation result is normalized and recorded
dem=dem./(1.01*max(abs(dem)));
audiowrite('res4.wav', dem, fs);

% Filtered demodulation result is normalized and recorded
dem=filter(fir1(255,4e3/(24e3)),1,dem);
dem=dem./(1.01*max(abs(dem)));
audiowrite('res5.wav', dem, fs);
```

B4. Analysis and Performance Evaluation

```
% Demod. performance analysis
% AM receiver analysis by Bilgehan ERKAL
% Karabuk 2020
clear all;

% stereo comparison file loading (48Khz stereo)
[iff1 , afs]=audioread('a2_aligned_st_Lres5_Ra2.wav');
[y1,~]=size(iff1);
% channel seperation and gain error correction
```

```
rec=1.0966*iff1(1:y1,1)';
a1=1*iff1(1:y1,2)';

% Calculate rms error and rms signal
diff=(a1-rec)/2;
err=(mean(diff.^2))^0.5;
a1_rms=(mean(a1.^2))^0.5;
fprintf('rms error: %d \nSNR(dB): %d \n', err, 20*log10(err/a1_rms));
audiowrite('diff.wav', diff, afs);
```

APPENDIX C.

VHDL CODE LISTINGS

C1. Top Module (cnt)

```
----------------------------------------------------------------------------------
-- Company: KARABUK UNIVERSITY
-- Engineer: Bilgehan ERKAL – Ali HANDER
--
-- Create Date:    14:07:31 04/21/2020
-- Design Name:    AM RX
-- Module Name:    cnt - Behavioral
-- Project Name:   AM RX
-- Target Devices: Spartan6-LX9
-- Tool versions:
-- Description: AM Receiver
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:

----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity cnt is
    Port ( SW                       : in  STD_LOGIC_VECTOR(3 downto
0);
           RX                       : in STD_LOGIC;--p4_1--green
           TX                       :    out    STD_LOGIC;--p4_2--
white
           LED                      : out  STD_LOGIC_VECTOR(7
downto 0);
           AUDIO                    : out STD_LOGIC;--p4_3
           AC97_SDO         :        out        STD_LOGIC;--p3_2--
mimas_p1_11
           AC97_SDI         :         in        STD_LOGIC;--p3_3--
mimas_p1_14
           AC97_BIT_CLK     :         in        STD_LOGIC;--p3_4--
mimas_p1_13
```

```
                    AC97_RESETN     :       out     STD_LOGIC;--p3_5--
mimas_p1_16
                    AC97_SYNC       :       out     STD_LOGIC;--p3_6--
mimas_p1_15
                    CLK_IN                  : in STD_LOGIC--100MHz on-
board clock
                   );
end cnt;

architecture Behavioral of cnt is

component pll2
port
 (-- Clock in ports
  CLK_IN1          : in     std_logic;
  -- Clock out ports
  CLK_OUT1         : out    std_logic;
  CLK_OUT2         : out    std_logic
 );
end component;

    COMPONENT AC97_ADAC
    PORT(
            AC97_int_SDI : IN std_logic;
            AC97_int_BIT_CLK : IN std_logic;
            DAC_L : IN std_logic_vector(17 downto 0);
            DAC_R : IN std_logic_vector(17 downto 0);
            clk_48 : IN std_logic;
            AC97_int_SDO : OUT std_logic;
            reset_n : IN std_logic;
            AC97_int_SYNC : OUT std_logic;
            ADC_L : OUT std_logic_vector(17 downto 0);
            ADC_R : OUT std_logic_vector(17 downto 0)
            );
    END COMPONENT;

COMPONENT sercomrx
    PORT(
            bin5            : OUT std_logic_vector(7 downto 0);
            bin4            : OUT std_logic_vector(7 downto 0);
            bin3            : OUT std_logic_vector(7 downto 0);
            bin2            : OUT std_logic_vector(7 downto 0);
            bin1            : OUT std_logic_vector(7 downto 0);
            bin0            : OUT std_logic_vector(7 downto 0);
            level    : OUT std_logic_vector(6 downto 0);
            data_valid: OUT std_logic;
            clk             : IN std_logic;
            serin    : IN std_logic;
            reset_n         : IN std_logic
```

```
        );
    END COMPONENT;

COMPONENT sercomtx
    PORT(
        bin5 : IN std_logic_vector(7 downto 0);
        bin4 : IN std_logic_vector(7 downto 0);
        bin3 : IN std_logic_vector(7 downto 0);
        bin2 : IN std_logic_vector(7 downto 0);
        bin1 : IN std_logic_vector(7 downto 0);
        bin0 : IN std_logic_vector(7 downto 0);
        data_valid : IN std_logic;
        clk : IN std_logic;
        reset_n : IN std_logic;
        binout5 : OUT std_logic_vector(7 downto 0);
        binout4 : OUT std_logic_vector(7 downto 0);
        binout3 : OUT std_logic_vector(7 downto 0);
        binout2 : OUT std_logic_vector(7 downto 0);
        binout1 : OUT std_logic_vector(7 downto 0);
        binout0 : OUT std_logic_vector(7 downto 0);
        serout : OUT std_logic
        );
END COMPONENT;

    COMPONENT frecalc
    PORT(
        digit_in : IN std_logic_vector(27 downto 0);
        reset_n : IN std_logic;
        data_valid : IN std_logic;
        clk : IN std_logic;
        phi_inc_out : OUT std_logic_vector(31 downto 0)
        );
    END COMPONENT;

    COMPONENT am_rx
    PORT(
        phi_inc         : IN std_logic_vector(31 downto 0);
        I_in            : IN std_logic_vector(15 downto 0);
        Q_in            : IN std_logic_vector(15 downto 0);
        s_sel           : IN std_logic_vector(3 downto 0);
        clk_48KHz : IN std_logic;
        clk             : IN std_logic;
        clip_indicator : out std_logic_vector(7 downto 0);
        I_out   : OUT std_logic_vector(15 downto 0);
        Q_out  : OUT std_logic_vector(15 downto 0)
        );
    END COMPONENT;

    COMPONENT dac16
```

```
    PORT(
            Clk : IN std_logic;
            Data : IN std_logic_vector(15 downto 0);
            PulseStream : OUT std_logic
            );
    END COMPONENT;

-- serial communication module signals
-- command completion stage indicator
signal stage : std_logic_vector(6 downto 0) := (others => '0');
signal sample  : std_logic := '0';
-- command data valid indicators
signal data_valid  : std_logic := '0';
-- completed command data
signal dat5 : std_logic_vector(7 downto 0) := (others => '0');
signal dat4 : std_logic_vector(7 downto 0) := (others => '0');
signal dat3 : std_logic_vector(7 downto 0) := (others => '0');
signal dat2 : std_logic_vector(7 downto 0) := (others => '0');
signal dat1 : std_logic_vector(7 downto 0) := (others => '0');
signal dat0 : std_logic_vector(7 downto 0) := (others => '0');
-- command data application digits indicating control frequency data
signal dig5 : std_logic_vector(7 downto 0) := (others => '0');
signal dig4 : std_logic_vector(7 downto 0) := (others => '0');
signal dig3 : std_logic_vector(7 downto 0) := (others => '0');
signal dig2 : std_logic_vector(7 downto 0) := (others => '0');
signal dig1 : std_logic_vector(7 downto 0) := (others => '0');
signal dig0 : std_logic_vector(7 downto 0) := (others => '0');
-- AM receiver module signals
-- phase increment value necessary to steer frequency of primary nco of am
receiver
signal phi_inc : std_logic_vector(31 downto 0) := (others => '0');
-- AM receiver output
signal I_out : std_logic_vector(15 downto 0) := (others => '0');
signal Q_out : std_logic_vector(15 downto 0) := (others => '0');
-- AM receiver input
signal I_in : std_logic_vector(15 downto 0) := (others => '0');
signal Q_in : std_logic_vector(15 downto 0) := (others => '0');
-- FPGA master reset signal
signal res_count : std_logic_vector(24 downto 0) := (others => '0');
signal reset_n : std_logic := '0';
-- ADAC (Audio card) reset signal
signal reset_n2 : std_logic := '0';
signal res_count2 : std_logic_vector(10 downto 0) := (others => '0');
-- clk_12288 12.288MHz clock live indicator
signal flash : std_logic_vector(22 downto 0) := (others => '0');
signal clk_count : std_logic_vector(8 downto 0) := (others => '0');
-- ADAC input and output signals
signal adata_L      : std_logic_vector(17 downto 0) := (others => '0');
signal adata_R      : std_logic_vector(17 downto 0) := (others => '0');
```

```
signal dacdata_L   : std_logic_vector(17 downto 0) := (others => '0');
signal dacdata_R   : std_logic_vector(17 downto 0) := (others => '0');
-- clipping indicator
signal c_ind                : std_logic_vector(7 downto 0);
--clock signals
signal clk                  : std_logic := '0';-- 36.684MHz master clock
signal clk_48KHz : std_logic := '0';-- 48KHz sampling rate clock
signal clk_12288  : STD_LOGIC;                  -- 12.288MHz ADAC master
clock
signal clk_6144    : STD_LOGIC := '0';-- 6.144MHz pll2 input clock
signal clk_36864  : STD_LOGIC;                  -- 36.684MHz master clock

begin

-- master module connectors
clk <= clk_36864;
AC97_RESETN <= reset_n2;
--rx--p4_1 --> tx pin of usb2serial cable (green)
--tx--p4_2 --> rx pin of usb2serial cable (white)
--gnd--(black)
--audio--p4_3
-- ADAC connectors
--ADC outputs
I_in <= adata_L(17 downto 2);
Q_in <= adata_R(17 downto 2);
--DAC inputs
dacdata_L <= I_out(15) & I_out(15 downto 0) & "0";
dacdata_R <= Q_out(15) & Q_out(15 downto 0) & "0";
-- led indicator connectors, uncomment necessary and
-- comment out unnecessary
--led(6 downto 0) <= stage(6 downto 0);--sercom completion levels
--led(7) <= flash(22);-- clock live indicator
led(7 downto 0) <= c_ind(7 downto 0);--clipping indicators

-- master clock generator
Inst_pll2 : pll2
  port map
   (-- Clock in ports
    CLK_IN1 => clk_6144,
    -- Clock out ports
    CLK_OUT1 => clk_36864,
    CLK_OUT2 => clk_12288
      );
-- pll2 input reference frequency (6.144MHz) derivator
-- master reference used is half of ADAC Bit clock at 12.288MHz
clk_6144_proc:process(AC97_BIT_CLK)
     begin
               if rising_edge(AC97_BIT_CLK) then
                        clk_6144 <= not clk_6144;
```

```
		end if;
	end process;

-- Sampling rate clock generator derived from 36.864MHz master clock
--36864/(384*2) = 48KHz
clk48KHz_proc:process(clk)
	begin
		if rising_edge(clk) then
			if reset_n = '1' then
				if clk_count = 383 then
					clk_48KHz <= not clk_48KHz;
					clk_count <= (others => '0');
				else
					clk_count <= clk_count + 1;
					clk_48KHz <= clk_48KHz;
				end if;
			else
				clk_count <= clk_count;
				clk_48KHz <= '0';
			end if;
		end if;
	end process;

-- Master reset of FPGA fabric
	reset_proc: process(AC97_BIT_CLK)
		begin
			if rising_edge(AC97_BIT_CLK) then
				if res_count(24) = '1' then
					res_count <= res_count;
				else
					res_count <= res_count + 1;
				end if;
			end if;
		end process;

	reset_n <= res_count(24);

-- ADAC reset generator, completed before master FPGA reset
-- It is solely derived from 100.00MHz FPGA master clock which is
-- the only live and stable clock before ADAC reset is completed
	AC97_reset_proc: process(clk_in)
		begin
			if rising_edge(clk_in) then
				if res_count2(10) = '1' then
					res_count2 <= res_count2;
				else
					res_count2 <= res_count2 + 1;
				end if;
			end if;
```

```
		end process;
	reset_n2 <= res_count2(10);

-- 12.288MHz clock live indicator
	flash_proc: process(clk_12288)
		begin
			if rising_edge(clk_12288) then
				flash <= not flash;
			end if;
		end process;

-- command data receive complete indicator
	sample_proc: process(clk)
		begin
			if rising_edge(clk) then
				if data_valid = '1' then
					sample <= not sample;
				else
					sample <= sample;
				end if;
			end if;
		end process;

-- ADAC module (AC97 soundcard module)
	Inst_AC97_ADAC: AC97_ADAC PORT MAP(
		AC97_int_SDO => AC97_SDO,
		AC97_int_SDI => AC97_SDI,
		AC97_int_BIT_CLK => AC97_BIT_CLK,
		reset_n => reset_n,
		AC97_int_SYNC => AC97_SYNC,
		DAC_L => dacdata_L,
		DAC_R => dacdata_R,
		ADC_L => adata_L,
		ADC_R => adata_R,
		clk_48 => clk_48KHz
	);

-- Serial communication receive module
-- This module is used to accept commands from PC at 9600bps
	Inst_sercomrx: sercomrx PORT MAP(
		bin5 => dat5,
		bin4 => dat4,
		bin3 => dat3,
		bin2 => dat2,
		bin1 => dat1,
		bin0 => dat0,
		level => stage,
		data_valid => data_valid,
		clk => clk,
```

```
            serin => rx,
            reset_n => reset_n
    );

-- Serial communication transmit module
-- This module is used to echo received commands to PC at 9600bps
    Inst_sercomtx: sercomtx PORT MAP(
            binout5 => dig5,
            binout4 => dig4,
            binout3 => dig3,
            binout2 => dig2,
            binout1 => dig1,
            binout0 => dig0,
            bin5 => dat5,
            bin4 => dat4,
            bin3 => dat3,
            bin2 => dat2,
            bin1 => dat1,
            bin0 => dat0,
            data_valid => data_valid,
            clk => clk,
            serout => tx,
            reset_n => reset_n
    );

-- Frequency calculation module
-- Takes frequency data which comes from PC as input
-- and calculates phase increment factor necessary to steer
-- primary nco frequency used in the AM RX module
Inst_frecalc: frecalc PORT MAP(
            digit_in => dig5 & dig4(3 downto 0) & dig3(3 downto 0) & dig2(3
downto 0) & dig1(3 downto 0) & dig0(3 downto 0),
            phi_inc_out => phi_inc,
            reset_n => reset_n,
            data_valid => data_valid,
            clk => clk
    );

-- Actual AM receiver module
    Inst_am_rx: am_rx PORT MAP(
            phi_inc => phi_inc,
            I_out => I_out,
            Q_out => Q_out,
            I_in => I_in,
            Q_in => Q_in,
            s_sel => SW,
            clip_indicator => c_ind,
            clk_48KHz => clk_48KHz,
            clk => clk
```

```
    );

-- Auxilary analog output port as Sigma-Delta DAC
    Inst_dac16: dac16 PORT MAP(
            Clk => clk,
            Data => not I_out(15) & I_out(14 downto 0),
            PulseStream => audio
    );
end Behavioral;
```

C2. LM4550 Soundcard Controller Module (AC97_ADAC)

```
----------------------------------------------------------------------------------

-- Company: KARABUK Un.
-- Engineer: Bilgehan ERKAL
--
-- Create Date:    11:26:16 04/28/2020
-- Design Name:
-- Module Name:    AC97_ADAC - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- Refer to LM4550 datasheet for details

----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity AC97_ADAC is
Port(
                    AC97_int_SDO            : out STD_LOGIC;
                    AC97_int_SDI            : in STD_LOGIC;
```

```
                AC97_int_BIT_CLK : in STD_LOGIC;
                reset_n                 : in STD_LOGIC;
                AC97_int_SYNC           : out STD_LOGIC;
                DAC_L                   :                   in
STD_LOGIC_VECTOR(17 downto 0);
                DAC_R                   :                   in
STD_LOGIC_VECTOR(17 downto 0);
                ADC_L                   :                   out
STD_LOGIC_VECTOR(17 downto 0);
                ADC_R                   :                   out
STD_LOGIC_VECTOR(17 downto 0);
                clk_48                  : in STD_LOGIC
);
end AC97_ADAC;

architecture Behavioral of AC97_ADAC is

COMPONENT comrom
  PORT (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(23 DOWNTO 0)
  );
END COMPONENT;

-- valid command present indicator signal
signal valbit                   : STD_LOGIC := '1';
-- output shift register for sending command and DAC data to soundcard
signal AC_97_out_sreg    : std_logic_vector(255 downto 0) := (others => '0');
-- input shift register for receiving command result data and ADC data
signal AC_97_in_sreg     : std_logic_vector(255 downto 0) := (others => '0');
-- command data signal
signal comrom_data              : std_logic_vector(23 downto 0) := (others =>
'0');
-- command data row signal used to address command rom
signal comrom_adr               : std_logic_vector(3 downto 0) := (others =>
'0');
-- DAC input registers
signal DAC_reg_L                : std_logic_vector(17 downto 0) :=
(others => '0');
signal DAC_reg_R                : std_logic_vector(17 downto 0) :=
(others => '0');
-- ADC output registers
signal ADC_reg_L                : std_logic_vector(17 downto 0) :=
(others => '0');
signal ADC_reg_R                : std_logic_vector(17 downto 0) :=
(others => '0');
-- preloaders used to resample module input before sending out to DAC
```

```
signal L_in                     : std_logic_vector(17 downto 0) := (others =>
'0');
signal R_in                     : std_logic_vector(17 downto 0) := (others =>
'0');
-- preregisters used to resample incoming ADC data before output by module
signal L_out                    : std_logic_vector(17 downto 0) := (others =>
'0');
signal R_out                    : std_logic_vector(17 downto 0) := (others =>
'0');
-- zero fill for output shift register
signal zero160                  : std_logic_vector(159 downto 0) := (others =>
'0');
-- synchronization signals
signal AC97_int_SYNC_reg : STD_LOGIC;
signal sync_count : std_logic_vector(7 downto 0) := (others => '0');

begin
AC97_int_SDO <= AC_97_out_sreg(255); -- DAC and command data output
AC97_int_SYNC <= AC97_int_SYNC_reg; -- soundcard sync input
-- connectors for ADC
ADC_L <= L_out;
ADC_R <= R_out;
-- Soundacard sync process
-- Generates a 48KHz sync clock necessary for soundcard using bit clock
AC97_SYNC_proc:process(AC97_int_BIT_CLK)
    begin
            if rising_edge(AC97_int_BIT_CLK) then
                    if reset_n = '1' then -- end of reset
                            sync_count <= sync_count + 1;
                            if sync_count = 0 then
                                    AC97_int_SYNC_reg <= '1';-- start of sync
impulse
                            else
                                    if sync_count = 16 then -- end of sync impulse
                                            AC97_int_SYNC_reg <= '0';
                                    else
                                            AC97_int_SYNC_reg                <=
AC97_int_SYNC_reg;-- other times, conserve status
                                    end if;
                            end if;
                    else -- reset conditions
                            sync_count <= (others => '0');
                            AC97_int_SYNC_reg <= '0';
                    end if;
            end if;
    end process;
-- command rom module
-- comrom holds initialization command data applied immediately after reset
Inst_comrom : comrom
```

```
 PORT MAP (
  clka => AC97_int_BIT_CLK,
  addra => comrom_adr,
  douta => comrom_data
 );

-- data and command data acquisition process
AC_97_ADAC_proc:process(AC97_int_BIT_CLK)
    begin
            if rising_edge(AC97_int_BIT_CLK) then
                    if reset_n = '1' then -- end of reset
                            -- serial data input from soundcard is shifted in to input
shift register (ADC data)
                            AC_97_in_sreg <= AC_97_in_sreg(254 downto 0) &
AC97_int_SDI;
                            -- start by the start of sync signal
                            if sync_count = 1 then
                                    -- reload output shift register with fresh
command data from comrom and DAC data from DAC data loading registers
                                    -- Slot0, Slot1, Slot2, Slot3-4 DAC data
                                    AC_97_out_sreg <= '1' & valbit & valbit &
"11000" & X"00" & comrom_data(23 downto 16) & X"000" & comrom_data(15
downto 0) & X"0" & DAC_reg_L(17 downto 0) & "00" & DAC_reg_R(17
downto 0) & "00" & zero160;
                            else
                                    AC_97_out_sreg <= AC_97_out_sreg(254
downto 0) & '0';-- other times animate shift register and send data to soundcard
                            end if;

                            -- Time to withdraw ADC data coming from soundcard
and loaded to input shift register
                            if sync_count = 2 then
                                    ADC_reg_L <= AC_97_in_sreg(199 downto
182);--slot3 data to ADC data input register Left channel
                                    ADC_reg_R <= AC_97_in_sreg(179 downto
162);--slot4 data to ADC data input register Right channel
                            else -- other times conserve ADC data input registers
                                    ADC_reg_L <= ADC_reg_L;
                                    ADC_reg_R <= ADC_reg_R;
                            end if;

                            -- Time to load preloading output registers with fresh
data (DAC and command data)
                            if sync_count = 3 then
                                    -- resampled module input loaded to preloading
output registers for DAC data
                                    DAC_reg_L <= L_in;
                                    DAC_reg_R <= R_in;
```

```
                                        -- if all the coomands listed in the command
rom is applied then enter wait state
                                        if comrom_adr = 5 then
                                                comrom_adr <= comrom_adr; -- wait at
the last command
                                        else
                                                comrom_adr <= comrom_adr + 1; --
proceed with the new command in the list
                                        end if;

                                        -- if last command is applied then mark
repeating last command as invalid by clearing the command valid bits
                                        if (comrom_adr = 4) or (comrom_adr = 5) then
                                                valbit <= '0'; -- invalid command
signaled
                                        else
                                                valbit <= '1'; -- valid command signaled
                                        end if;
                                else -- other times conserve status, enter wait state
                                        DAC_reg_L <= DAC_reg_L;
                                        DAC_reg_R <= DAC_reg_R;
                                        comrom_adr <= comrom_adr;
                                        valbit <= valbit;
                                end if;
                        else -- reset status
                                valbit <= '1';
                                AC_97_out_sreg <= (others => '0');
                                AC_97_in_sreg          <= (others => '0');
                                comrom_adr    <= (others => '0');
                                DAC_reg_L <= (others => '0');
                                DAC_reg_R <= (others => '0');
                                ADC_reg_L <= (others => '0');
                                ADC_reg_R <= (others => '0');
                                zero160 <= (others => '0');
                        end if;
                end if;
        end process;

-- Process for resampling input and output of ADAC module at 48KHz
AC97_sample_proc:process(clk_48)
        begin
                if rising_edge(clk_48) then
                        if reset_n = '1' then -- end of reset
                                L_in <= DAC_L;
                                R_in <= DAC_R;
                                L_out <= ADC_reg_L;
                                R_out <= ADC_reg_R;
                        else -- reset in order, clear registers to inital values
                                L_in <= (others => '0');
```

```
                                R_in <= (others => '0');
                                L_out <= (others => '0');
                                R_out <= (others => '0');
                            end if;
                    end if;
            end process;

    end Behavioral;
```

C3. Serial RX Module (Sercomrx)

```
    ----------------------------------------------------------------------------------

    -- Company:
    -- Engineer:
    --
    -- Create Date:    17:48:58 04/01/2020
    -- Design Name:
    -- Module Name:    sercomrx - Behavioral
    -- Project Name:
    -- Target Devices:
    -- Tool versions:
    -- Description:
    --
    -- Dependencies:
    --
    -- Revision:
    -- Revision 0.01 - File Created
    -- Additional Comments:

    ----------------------------------------------------------------------------------

    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;

    -- Uncomment the following library declaration if using
    -- arithmetic functions with Signed or Unsigned values
    --use IEEE.NUMERIC_STD.ALL;

    -- Uncomment the following library declaration if instantiating
    -- any Xilinx primitives in this code.
    --library UNISIM;
    --use UNISIM.VComponents.all;

    entity sercomrx is
    Port(
                bin5            : OUT std_logic_vector(7 downto 0);
                bin4            : OUT std_logic_vector(7 downto 0);
                bin3            : OUT std_logic_vector(7 downto 0);
                bin2            : OUT std_logic_vector(7 downto 0);
```

```
            bin1            : OUT std_logic_vector(7 downto 0);
            bin0            : OUT std_logic_vector(7 downto 0);
            level   : OUT std_logic_vector(6 downto 0);
            data_valid: OUT std_logic;
            clk             : IN std_logic;
            serin   : IN std_logic;
            reset_n         : IN std_logic

);
end sercomrx;

architecture Behavioral of sercomrx is

--Received data register
signal data : std_logic_vector(7 downto 0) := (others => '0');
--Received command completion data
signal stage : std_logic_vector(6 downto 0) := (others => '0');
--Received data bins in ASCII form
signal dat5 : std_logic_vector(7 downto 0) := (others => '0');--Sign (+/-)
signal dat4 : std_logic_vector(7 downto 0) := (others => '0');--MSB
signal dat3 : std_logic_vector(7 downto 0) := (others => '0');
signal dat2 : std_logic_vector(7 downto 0) := (others => '0');
signal dat1 : std_logic_vector(7 downto 0) := (others => '0');
signal dat0 : std_logic_vector(7 downto 0) := (others => '0');--LSB
--Received bins counter
signal dat_count : std_logic_vector(2 downto 0) := (others => '0');
--Serial input resample shift register
--Input is resampled 4 times in order to catch bits appropriately (10bits *4 =
40bits)
signal bits : std_logic_vector(39 downto 0) := (others => '1');
--resampling clock divider
signal counter : std_logic_vector(9 downto 0) := (others => '0');
--Received byte ok signal
signal byte_ok : std_logic := '0';
--Received byte ok signal delay registers
signal dat_ok1 : std_logic := '0';
signal dat_ok  : std_logic := '0';
--Data valid signal, When dat_valid is active there is new and valid data at data
bin outputs
signal blink  : std_logic := '0';
signal blink_del  : std_logic := '0';
signal dat_valid  : std_logic := '0';

begin
--calculate and generate byte ok signal (indicates that valid start-bit- 8-bit data
and stop-bit sequence in the bits resampling register)
    byte_ok <= -- start-bit "0" bits(2:1) + LSB first, MSB last data bits (0:7) +
stopbit "0" bits(38:37)
```

```
                                        (bits(38) and bits(37)) and (bits(34) xnor
bits(33)) and (bits(30) xnor bits(29)) and (bits(26) xnor bits(25)) and
                                (bits(22) xnor bits(21)) and (bits(18) xnor bits(17)) and
(bits(14) xnor bits(13)) and (bits(10) xnor bits(9)) and
                                (bits(6) xnor bits(5)) and (bits(2) nor bits(1));
--module connectors
    bin5 <= dat5;
    bin4 <= dat4;
    bin3 <= dat3;
    bin2 <= dat2;
    bin1 <= dat1;
    bin0 <= dat0;
    data_valid <= dat_valid;
    level <= stage;
--serial communication process
    ser_comm: process(clk)--ticks at 36864MHz
            begin
                    if rising_edge(clk) then
                            if reset_n = '1' then
                                    if counter = 959 then --resampling clock at
36864/960 = 4* 9600 = 38400
                                            bits <= serin & bits(39 downto 1);--
resample serial input, data comes as LSB first
                                            counter <= (others => '0');--reset
counter
                                    else
                                            counter <= counter+1;--wait till next
sample
                                    end if;
                                    if byte_ok = '1' then-- there is valid data in bits
resample register transfer it to data register
                                            data <= bits(34) & bits(30) & bits(26) &
bits(22) & bits(18) & bits(14) & bits(10) & bits(6);
                                            bits <= (others => '1');--setup bits
register for next new data
                                    end if;
                            else--reset in order
                                    data(7 downto 0) <= (others => '0');
                                    bits(39 downto 0) <= (others => '1');
                                    counter(9 downto 0) <= (others => '0');
                            end if;
                    end if;
            end process;
--byte_ok signal delay process
    dat_ok_proc: process(clk)
            begin
                    if rising_edge(clk) then
                            dat_ok1 <= byte_ok;
                            dat_ok <= dat_ok1;
```

```
                end if;
        end process;
--transfer valid data to apropriate data bin process
-- a FSM is used to track the next data bin to reload
-- if data is not valid for the actual data bin
-- all data located in the data bins so far is discarded
-- and the process starts from the beginning
    dat_xfer_proc: process(clk)
        begin
            if rising_edge(clk) then
                if reset_n = '1' then
                    if dat_ok = '1' then
                        CASE dat_count(2 downto 0) IS
                            WHEN "000" => --test data for f-command
                                dat5 <= dat5;
                                dat4 <= dat4;
                                dat3 <= dat3;
                                dat2 <= dat2;
                                dat1 <= dat1;
                                dat0 <= dat0;
                                blink <= blink;
                                if data = 102 then--character "f" is received, next incoming data must be sign(+ or -)
                                    stage(0) <= '1';--first stage is completed successfully
                                    dat_count <= dat_count + 1;--proceed with next data
                                else
                                    dat_count <= (others => '0');-- received data is not valid (other than "f" character), start from beginning
                                    stage <= (others => '0');-- clear all stages
                                end if;
                            WHEN "001" => --test data for sign
                                dat4 <= dat4;
                                dat3 <= dat3;
                                dat2 <= dat2;
                                dat1 <= dat1;
                                dat0 <= dat0;
                                blink <= blink;
                                if ((data = 43) or (data = 45)) then--data is plus or minus character
                                    stage(1) <= '1';
                                    dat5 <= data;--record it in first data bin from left
```

```
                                        dat_count <=
dat_count + 1;
                                    else--start all over again
                                        dat_count <=
(others => '0');
                                        stage <= (others
=> '0');
                                        dat5 <= dat5;

                                    end if;
                                WHEN "010" =>--test for msb
(it must be either a "0", "1" or "2")
                                        dat5 <= dat5;
                                        dat3 <= dat3;
                                        dat2 <= dat2;
                                        dat1 <= dat1;
                                        dat0 <= dat0;
                                        blink <= blink;
                                    if ((data > 47) and (data
< 51)) then--it is a "0", "1" or "2"
                                        stage(2) <= '1';
                                        dat4 <= data;--
then register it
                                        dat_count <=
dat_count + 1;
                                    else--start all over again
                                        dat_count <=
(others => '0');
                                        stage <= (others
=> '0');
                                        dat4 <= dat4;
                                    end if;
                                WHEN "011" =>--test for
second msb (it must be either a "0", "1", "2" or "3" if first msb is "2" otherwise 0-
9)
                                        dat5 <= dat5;
                                        dat4 <= dat4;
                                        dat2 <= dat2;
                                        dat1 <= dat1;
                                        dat0 <= dat0;
                                        blink <= blink;
                                    if dat4 = 50 then--first
msb is "2"
                                        if ((data > 47)
and (data < 52)) then
                                            stage(3)
<= '1';
                                            dat3 <=
data;
```

```
                                        dat_count
<= dat_count + 1;
                                    else--start all over
again
                                        dat_count
<= (others => '0');
                                        stage   <=
(others => '0');
                                        dat3    <=
dat3;
                                    end if;
                                else-- first msb is 0 or 1
                                    if  ((data  >  47)
and (data < 58)) then
                                        stage(3)
<= '1';
                                        dat3    <=
data;
                                        dat_count
<= dat_count + 1;
                                    else--start all over
again
                                        dat_count
<= (others => '0');
                                        stage   <=
(others => '0');
                                        dat3    <=
dat3;
                                    end if;
                                end if;
                            WHEN "100" =>--third msb (it
can be 0-9)
                                    dat5 <= dat5;
                                    dat4 <= dat4;
                                    dat3 <= dat3;
                                    dat1 <= dat1;
                                    dat0 <= dat0;
                                    blink <= blink;
                                if ((data > 47) and (data
< 58)) then
                                    stage(4) <= '1';
                                    dat2 <= data;
                                    dat_count       <=
dat_count + 1;
                                else--start all over again
                                    dat_count       <=
(others => '0');
                                    stage  <=  (others
=> '0');
```

```
                dat2 <= dat2;
            end if;
        WHEN "101" =>-- fourth msb
(0-9)
                dat5 <= dat5;
                dat4 <= dat4;
                dat3 <= dat3;
                dat2 <= dat2;
                dat0 <= dat0;
                blink <= blink;
            if ((data > 47) and (data
< 58)) then
                stage(5) <= '1';
                dat1 <= data;
                dat_count       <=
dat_count + 1;
            else--start all over again
                dat_count       <=
(others => '0');
                stage  <=  (others
=> '0');
                dat1 <= dat1;
            end if;
        WHEN "110" =>--lsb (0-9)
                dat5 <= dat5;
                dat4 <= dat4;
                dat3 <= dat3;
                dat2 <= dat2;
                dat1 <= dat1;
            dat_count  <=  (others  =>
'0');--start at the beginning for new data
            if ((data > 47) and (data
< 58)) then
                stage(6) <= '1';
                dat0 <= data;
                blink    <=    not
blink;--changeover blink
            else--start all over again
                stage  <=  (others
=> '0');
                blink <= blink;
                dat0 <= dat0;
            end if;
        WHEN  OTHERS  =>--start  all
over again
                dat_count       <=
(others => '0');
                dat5 <= dat5;
                dat4 <= dat4;
```

```
                                                dat3 <= dat3;
                                                dat2 <= dat2;
                                                dat1 <= dat1;
                                                dat0 <= dat0;
                                                blink <= blink;
                                                stage  <=  (others
    => '0');
                                                END CASE;
                                        else-- there is no valid new data so wait for one
    to come
                                                dat_count <= dat_count;
                                                dat5 <= dat5;
                                                dat4 <= dat4;
                                                dat3 <= dat3;
                                                dat2 <= dat2;
                                                dat1 <= dat1;
                                                dat0 <= dat0;
                                                blink <= blink;
                                                stage <= stage;
                                        end if;
                                else-- reset in order
                                        dat_count <= (others => '0');
                                        dat5 <= (others => '0');
                                        dat4 <= (others => '0');
                                        dat3 <= (others => '0');
                                        dat2 <= (others => '0');
                                        dat1 <= (others => '0');
                                        dat0 <= (others => '0');
                                        blink <= '0';
                                        stage <= (others => '0');
                                end if;
                        end if;
                end process;
    -- data valid signal process
        dat_valid_proc: process(clk)
                begin
                        if rising_edge(clk) then
                                if reset_n = '1' then
                                        dat_valid  <=  blink  xor  blink_del;--there  is
    changeover in blink so there is new valid data
                                        blink_del <= blink;-- delay blink signal so that
    data valid signal is active for only one clock cycle
                                else--reset in order
                                        dat_valid <= '0';
                                        blink_del <= '0';
                                end if;
                        end if;
                end process;
end Behavioral;
```

C4. Serial TX Module (Sercomtx)

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    17:48:58 04/01/2020
-- Design Name:
-- Module Name:    sercomrx - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:

----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity sercomtx is
Port(
            binout5             : OUT std_logic_vector(7 downto 0);
            binout4             : OUT std_logic_vector(7 downto 0);
            binout3             : OUT std_logic_vector(7 downto 0);
            binout2             : OUT std_logic_vector(7 downto 0);
            binout1             : OUT std_logic_vector(7 downto 0);
            binout0             : OUT std_logic_vector(7 downto 0);
            bin5        : IN std_logic_vector(7 downto 0);
            bin4        : IN std_logic_vector(7 downto 0);
            bin3        : IN std_logic_vector(7 downto 0);
            bin2        : IN std_logic_vector(7 downto 0);
            bin1        : IN std_logic_vector(7 downto 0);
            bin0        : IN std_logic_vector(7 downto 0);
            data_valid: IN std_logic;
```

```
            clk             : IN std_logic;
            serout  : OUT std_logic;
            reset_n         : IN std_logic

);
end sercomtx;

architecture Behavioral of sercomtx is

COMPONENT text_rom
  PORT (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END COMPONENT;
-- output digit data registers
signal dig5 : std_logic_vector(7 downto 0) := (others => '0');
signal dig4 : std_logic_vector(7 downto 0) := (others => '0');
signal dig3 : std_logic_vector(7 downto 0) := (others => '0');
signal dig2 : std_logic_vector(7 downto 0) := (others => '0');
signal dig1 : std_logic_vector(7 downto 0) := (others => '0');
signal dig0 : std_logic_vector(7 downto 0) := (others => '0');
-- serial data transfer registers
signal busyshiftreg : std_logic_vector(9 downto 0) := (others => '0');
signal datashiftreg : std_logic_vector(9 downto 0) := (others => '1');
-- clock divider counter
signal txcounter : std_logic_vector(12 downto 0) := (others => '0');
-- data register, reloading register for serial data transfer register
signal data : std_logic_vector(7 downto 0) := (others => '0');
-- caption text rom signals
signal rom_counter : std_logic_vector(7 downto 0) := (others => '0');
signal douta : std_logic_vector(7 downto 0) := (others => '0');
-- internal control signals
signal state_counter : std_logic_vector(3 downto 0) := (others => '0');
signal timer : std_logic_vector(1 downto 0) := (others => '0');
signal valid_data : std_logic := '0';
signal data2send : std_logic := '0';

begin
--caption text rom
Inst_romtext : text_rom
  PORT MAP (
    clka => clk,
    addra => rom_counter(5 downto 0),
    douta => douta
  );
 -- module connectors
      serout <= datashiftreg(0);--data shifted as LSB first MSB last
```

```
    data2send <= (valid_data);
    binout5 <= dig5;
    binout4 <= dig4;
    binout3 <= dig3;
    binout2 <= dig2;
    binout1 <= dig1;
    binout0 <= dig0;
-- main serial communication process
    ser_comm_tx: process(clk)
            begin
                    if rising_edge(clk) then
                            if reset_n = '1' then
                                    if data2send = '1' then--there is data waiting to
be sent
                                            if busyshiftreg(0) = '0' then--sender is
not busy then reload new data from data register to data shift register
                                                    busyshiftreg <= (others => '1');--
set busy signal to prevent unintended reloading of data shift register
                                                    txcounter <= (others => '0');--
clear tx counter for the timing of new transfer
                                                    datashiftreg <= '1' & data & '0';--
reload datashift register with fresh data and also include start and stop bits
                                            else
                                                    if txcounter = 3839 then--bit
clock = 36864/3840 = 9.6kbps
                                                            datashiftreg <= '1' &
datashiftreg(9 downto 1);--time to shift out a new bit
                                                            busyshiftreg <= '0' &
busyshiftreg(9 downto 1);--count sent bits when complete busy signal is made
inactive automatically
                                                            txcounter <= (others =>
'0');-- clear clock divider
                                                    else--wait till next bit to out
                                                            txcounter
<=
txcounter+1;
                                                            datashiftreg
<=
datashiftreg;
                                                            busyshiftreg
<=
busyshiftreg;
                                                    end if;
                                            end if;
                                    else--there is no new data so wait in ready state
(not busy)
                                            busyshiftreg <= (others => '0');
                                            txcounter <= (others => '0');
                                            datashiftreg <= (others => '1');

                                    end if;
                            else-- reset in order
```

```
                busyshiftreg <= (others => '0');
                txcounter <= (others => '0');
                datashiftreg <= (others => '1');
            end if;
        end if;
    end process;
-- outgoing data reloading process
    data_proc: process(clk)
        begin
            if rising_edge(clk) then
                if reset_n = '1' then
                    CASE state_counter(3 downto 0) IS
                        WHEN "0000" =>-- caption text state, withdraw text data from rom
                            if (rom_counter = 64) and (busyshiftreg(0) = '0') then-- end of rom data so proceed with sending out incoming data (echo received data)
                                state_counter <= state_counter + 1;
                                valid_data <= '0';
                                rom_counter <= rom_counter;
                                data <= data;
                            else--withdraw text data from rom
                                state_counter <= state_counter;
                                valid_data <= '1';
                                if busyshiftreg(0) = '0' then-- serial transmitter is ready for new data
                                    rom_counter <= rom_counter + 1;--proceed with next line
                                    data <= douta;--reload new data
                                else--serial xmitter is busy so wait until not busy
                                    rom_counter <= rom_counter;
                                    data <= data;
                                end if;
                            end if;
                        WHEN "0001" =>--xmit sign of incoming data
                            data <= dig5;
                            if (timer = "11") and (busyshiftreg(0) = '0') then--data accepted for xmit so proceed next state and wait till xmitter is ready to accept new data
                                state_counter <= state_counter + 1;
```

```
                        valid_data <= '0';
                        timer <= "00";
                    else--xmitter is not ready so wait
                        state_counter <=
state_counter;
                        valid_data <= '1';
                        if timer = "11" then
                            timer <= timer;
                        else
                            timer <= timer +
1;
                        end if;
                    end if;
                WHEN "0010" =>--xmit first msb of
incoming data
                    data <= dig4;
                    if (timer = "11") and
(busyshiftreg(0) = '0') then
                        state_counter <=
state_counter + 1;
                        valid_data <= '0';
                        timer <= "00";
                    else
                        state_counter <=
state_counter;
                        valid_data <= '1';
                        if timer = "11" then
                            timer <= timer;
                        else
                            timer <= timer +
1;
                        end if;
                    end if;
                WHEN "0011" =>--xmit second msb
                    data <= dig3;
                    if (timer = "11") and
(busyshiftreg(0) = '0') then
                        state_counter <=
state_counter + 1;
                        valid_data <= '0';
                        timer <= "00";
                    else
                        state_counter <=
state_counter;
                        valid_data <= '1';
                        if timer = "11" then
                            timer <= timer;
                        else
```

```
                                        timer <= timer +
1;
                                    end if;
                                end if;
                            WHEN "0100" =>--xmit third msb
                                data <= dig2;
                                if (timer = "11") and
(busyshiftreg(0) = '0') then
                                    state_counter <=
state_counter + 1;
                                    valid_data <= '0';
                                    timer <= "00";
                                else
                                    state_counter <=
state_counter;
                                    valid_data <= '1';
                                    if timer = "11" then
                                        timer <= timer;
                                    else
                                        timer <= timer +
1;
                                    end if;
                                end if;
                            WHEN "0101" =>--xmit fourth msb
                                data <= dig1;
                                if (timer = "11") and
(busyshiftreg(0) = '0') then
                                    state_counter <=
state_counter + 1;
                                    valid_data <= '0';
                                    timer <= "00";
                                else
                                    state_counter <=
state_counter;
                                    valid_data <= '1';
                                    if timer = "11" then
                                        timer <= timer;
                                    else
                                        timer <= timer +
1;
                                    end if;
                                end if;
                            WHEN "0110" =>--xmit lsb
                                data <= dig0;
                                if (timer = "11") and
(busyshiftreg(0) = '0') then
                                    state_counter <=
state_counter + 1;
                                    valid_data <= '0';
```

```
                    timer <= "00";
                else
                    state_counter <=
state_counter;
                    valid_data <= '1';
                    if timer = "11" then
                        timer <= timer;
                    else
                        timer <= timer +
1;
                    end if;
                end if;
            WHEN "0111" =>--xmit CR
                data <= X"0D";
                if (timer = "11") and
(busyshiftreg(0) = '0') then
                    state_counter <=
state_counter + 1;
                    valid_data <= '0';
                    timer <= "00";
                else
                    state_counter <=
state_counter;
                    valid_data <= '1';
                    if timer = "11" then
                        timer <= timer;
                    else
                        timer <= timer +
1;
                    end if;
                end if;
            WHEN "1000" =>--xmit LF so proceed
new data with new line
                data <= X"0A";
                if (timer = "11") and
(busyshiftreg(0) = '0') then
                    state_counter <=
state_counter + 1;
                    valid_data <= '0';
                    timer <= "00";
                else
                    state_counter <=
state_counter;
                    valid_data <= '1';
                    if timer = "11" then
                        timer <= timer;
                    else
                        timer <= timer +
1;
```

```
                                                end if;
                                            end if;
                                        WHEN OTHERS =>--start from sending sign of new data (echo incoming data forever as there is new valid data)
                                                if data_valid = '1' then--there is new valid data waiting to be echoed out
                                                        state_counter <= "0001";
                                                else--wait for new data
                                                        state_counter <= state_counter;
                                                end if;
                                                timer <= "00";
                                                valid_data <= '0';
                                                rom_counter <= rom_counter;
                                                data <= data;
                                        END CASE;
                                else--reset in order
                                        valid_data <= '0';
                                        rom_counter <= (others => '0');
                                        state_counter <= (others => '0');
                                        timer <= (others => '0');
                                        data <= X"61";
                                end if;
                        end if;
                end process;
-- sample incoming data process
    update_digit_proc: process(clk)
                begin
                        if rising_edge(clk) then
                                if reset_n = '1' then
                                        if data_valid = '1' then--there is valid data waiting to be echoed
                                                dig5 <= bin5;
                                                dig4 <= bin4;
                                                dig3 <= bin3;
                                                dig2 <= bin2;
                                                dig1 <= bin1;
                                                dig0 <= bin0;
                                        else--wait
                                                dig5 <= dig5;
                                                dig4 <= dig4;
                                                dig3 <= dig3;
                                                dig2 <= dig2;
                                                dig1 <= dig1;
                                                dig0 <= dig0;
                                        end if;
                                else--reset state (initial value -15000)
```

```
                                    dig5 <= X"2D";
                                    dig4 <= X"31";
                                    dig3 <= X"35";
                                    dig2 <= X"30";
                                    dig1 <= X"30";
                                    dig0 <= X"30";
                                end if;
                            end if;
                        end process;

end Behavioral;
```

C5. Phase Increment Calculator Module (frecalc)

```
    ----------------------------------------------------------------------------

    -- Company:
    -- Engineer:
    --
    -- Create Date:    17:46:16 04/28/2020
    -- Design Name:
    -- Module Name:    frecalc - Behavioral
    -- Project Name:
    -- Target Devices:
    -- Tool versions:
    -- Description:
    --
    -- Dependencies:
    --
    -- Revision:
    -- Revision 0.01 - File Created
    -- Additional Comments:
    ----------------------------------------------------------------------------
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;

    -- Uncomment the following library declaration if using
    -- arithmetic functions with Signed or Unsigned values
    use IEEE.NUMERIC_STD.ALL;

    -- Uncomment the following library declaration if instantiating
    -- any Xilinx primitives in this code.
    --library UNISIM;
    --use UNISIM.VComponents.all;

    entity frecalc is
    Port(
    digit_in                : in STD_LOGIC_VECTOR(27 downto 0);
    phi_inc_out             : out std_logic_vector(31 downto 0) := (others => '0');
```

```
reset_n                         : in STD_LOGIC;
data_valid              : in STD_LOGIC;
clk                             : in std_logic

);
end frecalc;

architecture Behavioral of frecalc is

COMPONENT FREQ
    PORT(
            bin : IN std_logic_vector(27 downto 0);
            pinc : OUT std_logic_vector(31 downto 0)
            );
END COMPONENT;

signal phi_inc : std_logic_vector(31 downto 0) := (others => '0');--calculated
phase increment
signal freq_counter : std_logic_vector(7 downto 0) := (others => '0');--wait
counter
signal pinc : std_logic_vector(31 downto 0) := (others => '0');--precalculation
register
signal bin : STD_LOGIC_VECTOR(27 downto 0);--input frequency data in
BCD format + sign data

begin
phi_inc_out <= phi_inc;

    Inst_FREQ: FREQ PORT MAP(
            bin => digit_in,
            pinc => pinc
    );
    freq_proc:process(clk)
            begin
                    if rising_edge(clk) then
                            if reset_n = '1' then
                                    if freq_counter = X"00" then--wait for new data
                                            phi_inc <= phi_inc;
                                            if data_valid = '1' then--there is new
data so start calculation
                                                    freq_counter <= freq_counter +
1;
                                            else
                                                    freq_counter <= freq_counter;
                                            end if;
                                    else-- calculation in progress
                                            freq_counter <= freq_counter + 1;
                                            if freq_counter = X"FF" then--
calculation is complete
```

```
                                        phi_inc <= pinc;--update phase
    increment output with new one
                                    else--wait till calculation is complete
                                        phi_inc <= phi_inc;
                                    end if;
                                end if;
                            else--reset state
                                freq_counter <= X"01";
                                phi_inc <= (others => '0');

                            end if;
                        end if;
                end process;

end Behavioral;
```

C6. AM Receiver Module (am_rx)

```
    ----------------------------------------------------------------------------------
    -- Company: KARABUK UNIVERSITY
    -- Engineer: Bilgehan ERKAL -Ali HANDER
    --
    -- Create Date:    18:24:23 04/28/2020
    -- Design Name:
    -- Module Name:    am_rx - Behavioral
    -- Project Name:
    -- Target Devices:
    -- Tool versions:
    -- Description:
    --
    -- Dependencies:
    --
    -- Revision:
    -- Revision 0.01 - File Created
    -- Additional Comments:
    ----------------------------------------------------------------------------------
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;

    -- Uncomment the following library declaration if using
    -- arithmetic functions with Signed or Unsigned values
    use IEEE.NUMERIC_STD.ALL;

    -- Uncomment the following library declaration if instantiating
    -- any Xilinx primitives in this code.
    --library UNISIM;
    --use UNISIM.VComponents.all;
```

```
entity am_rx is
   Port (
      phi_inc : in std_logic_vector(31 downto 0);
      I_out   : out std_logic_vector(15 downto 0);
      Q_out           : out std_logic_vector(15 downto 0);
      I_in    : in std_logic_vector(15 downto 0);
      Q_in    : in std_logic_vector(15 downto 0);
      s_sel   : in std_logic_vector(3 downto 0);
      clip_indicator : out std_logic_vector(7 downto 0);
      clk_48KHz : in  STD_LOGIC;
      clk             : in  STD_LOGIC

      );
end am_rx;

architecture Behavioral of am_rx is

     COMPONENT clipping_indicator_multi_channel
     PORT(
               clip_in_I : IN std_logic_vector(15 downto 0);
               clip_in_Q : IN std_logic_vector(15 downto 0);
               clk : IN std_logic;
               clip_out : OUT std_logic
               );
     END COMPONENT;

component comp_mult1
     port (
     ar: in std_logic_vector(15 downto 0);
     ai: in std_logic_vector(15 downto 0);
     br: in std_logic_vector(15 downto 0);
     bi: in std_logic_vector(15 downto 0);
     clk: in std_logic;
     pr: out std_logic_vector(32 downto 0);
     pi: out std_logic_vector(32 downto 0));
end component;

COMPONENT nco1
  PORT (
    clk : IN STD_LOGIC;
    pinc_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    cosine : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    sine : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END COMPONENT;

component fir1
     port (
     clk: in std_logic;
```

```
      rfd: out std_logic;
      rdy: out std_logic;
      din: in std_logic_vector(15 downto 0);
      dout: out std_logic_vector(34 downto 0));
end component;

COMPONENT nco2
  PORT (
    clk : IN STD_LOGIC;
    cosine : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    sine : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END COMPONENT;

COMPONENT mult
  PORT (
    clk : IN STD_LOGIC;
    a : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    b : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    p : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;

-- first complex multiplier and nco signals
signal cos : std_logic_vector(15 downto 0) := (others => '0');
signal sin : std_logic_vector(15 downto 0) := (others => '0');
signal cmult_Iout1 : std_logic_vector(32 downto 0) := (others => '0');
signal cmult_Qout1 : std_logic_vector(32 downto 0) := (others => '0');

-- first LPF signals
signal lpf_I_din : std_logic_vector(15 downto 0) := (others => '0');
signal lpf_Q_din : std_logic_vector(15 downto 0) := (others => '0');
signal lpf_I_dout : std_logic_vector(34 downto 0) := (others => '0');
signal lpf_Q_dout : std_logic_vector(34 downto 0) := (others => '0');

-- second complex multiplier and nco signals
signal I_in2 : std_logic_vector(15 downto 0) := (others => '0');
signal Q_in2 : std_logic_vector(15 downto 0) := (others => '0');
signal cos2 : std_logic_vector(15 downto 0) := (others => '0');
signal sin2 : std_logic_vector(15 downto 0) := (others => '0');
signal cmult_Iout2 : std_logic_vector(32 downto 0) := (others => '0');
signal cmult_Qout2 : std_logic_vector(32 downto 0) := (others => '0');

-- realizer circuit signals
signal adder_I_in : std_logic_vector(15 downto 0) := (others => '0');
signal adder_Q_in : std_logic_vector(15 downto 0) := (others => '0');
signal adder_out : std_logic_vector(15 downto 0) := (others => '0');

-- Squaring type AM demodulator signals
```

```
signal squarer_in : std_logic_vector(15 downto 0) := (others => '0');
signal squarer_out : std_logic_vector(31 downto 0) := (others => '0');

-- Output LPF signals
signal lpf_out_din : std_logic_vector(15 downto 0) := (others => '0');
signal lpf_out_dout : std_logic_vector(34 downto 0) := (others => '0');

-- module outputs
signal I_out_x : std_logic_vector(15 downto 0) := (others => '0');
signal Q_out_x : std_logic_vector(15 downto 0) := (others => '0');

begin

-- intercomponent coarse level adjustment connectors
-- First LPF inputs
lpf_I_din <= cmult_Iout1(32) & cmult_Iout1(28 downto 14);
lpf_Q_din <= cmult_Qout1(32) & cmult_Qout1(28 downto 14);
--Second complex multiplier inputs
I_in2 <= lpf_I_dout(34) & lpf_I_dout(31 downto 17);
Q_in2 <= lpf_Q_dout(34) & lpf_Q_dout(31 downto 17);
--Realizer inputs
adder_I_in <= cmult_Iout2(32) & cmult_Iout2(28 downto 14);
adder_Q_in <= cmult_Qout2(32) & cmult_Qout2(28 downto 14);
-- AM Demodulator input
squarer_in <= adder_out(15) & adder_out(14 downto 0);
-- Output (final) LPF inputs
lpf_out_din <= squarer_out(31) & squarer_out(30 downto 16);
-- Module outputs (Despite 2-ch complex it is real in fact)
I_out_x <= lpf_out_dout(34) & lpf_out_dout(30 downto 16);
Q_out_x <= lpf_out_dout(34) & lpf_out_dout(30 downto 16);

-- Source selector process
s_sel_proc: process(clk)
    begin
        if rising_edge(clk) then
            CASE s_sel(3 downto 0) IS
                WHEN "1111" =>-- Demodulated final output
                    I_out <= I_out_x;
                    Q_out <= Q_out_x;

                WHEN "1110" =>-- Module input (first complex
multiplier input)
                    I_out <= I_in;
                    Q_out <= Q_in;

                WHEN "1101" =>-- Fist complex multiplier output,
First LPF input
                    I_out <= lpf_I_din;
                    Q_out <= lpf_Q_din;
```

```
                        WHEN "1011" =>-- First LPF output, second complex
multiplier input
                                I_out <= I_in2;
                                Q_out <= Q_in2;

                        WHEN "0111" =>-- Second complex multiplier output,
realizer input
                                I_out <= adder_I_in;
                                Q_out <= adder_Q_in;

                        WHEN "1100" =>-- Realizer output, AM demodulator
input
                                I_out <= squarer_in;
                                Q_out <= squarer_in;

                        WHEN "1010" =>-- AM demodulator output, Output
LPF input
                                I_out <= lpf_out_din;
                                Q_out <= lpf_out_din;

                        WHEN "0110" =>-- Output LPF output, module output
                                I_out <= I_out_x;
                                Q_out <= Q_out_x;

                        WHEN OTHERS =>-- module output
                                I_out <= I_out_x;
                                Q_out <= Q_out_x;

                END CASE;
        end if;
    end process;
-- Module input (first complex multiplier input) clipping indicator
cind0: clipping_indicator_multi_channel PORT MAP(
        clip_in_I => I_in,
        clip_in_Q => Q_in,
        clip_out => clip_indicator(0),
        clk => clk
    );
-- First LPF input clipping indicator
cind1: clipping_indicator_multi_channel PORT MAP(
        clip_in_I => lpf_I_din,
        clip_in_Q => lpf_Q_din,
        clip_out => clip_indicator(1),
        clk => clk
    );
-- Second complex multiplier input clipping indicator
cind2: clipping_indicator_multi_channel PORT MAP(
        clip_in_I => I_in2,
```

```
                clip_in_Q => Q_in2,
                clip_out => clip_indicator(2),
                clk => clk
    );
-- Realizer input clipping indicator
cind3: clipping_indicator_multi_channel PORT MAP(
                clip_in_I => adder_I_in,
                clip_in_Q => adder_Q_in,
                clip_out => clip_indicator(3),
                clk => clk
    );
-- AM demodulator input clipping indicator
cind4: clipping_indicator_multi_channel PORT MAP(
                clip_in_I => squarer_in,
                clip_in_Q => squarer_in,
                clip_out => clip_indicator(4),
                clk => clk
    );
-- Output LPF input clipping indicator
cind5: clipping_indicator_multi_channel PORT MAP(
                clip_in_I => lpf_out_din,
                clip_in_Q => lpf_out_din,
                clip_out => clip_indicator(5),
                clk => clk
    );
-- Module output (Output LPF output) clipping indicator
cind6: clipping_indicator_multi_channel PORT MAP(
                clip_in_I => I_out_x,
                clip_in_Q => Q_out_x,
                clip_out => clip_indicator(6),
                clk => clk
    );
-- Empty indicator (reserved for future use)
clip_indicator(7) <= '0';

-- First complex multiplier
Inst_comp_mult1 : comp_mult1
                port map (
                        ar => I_in,
                        ai => Q_in,
                        br => cos,--x"7FFF",--cos,--
                        bi => sin,--x"0000",--sin,--
                        clk => clk_48KHz,
                        pr => cmult_Iout1,
                        pi => cmult_Qout1
                );

-- First NCO (steered by Phase increment value calculated from PC input
frequency data)
```

```
Inst_nco1 : nco1
  PORT MAP (
    clk => clk_48KHz,
    pinc_in => phi_inc,
    cosine => cos,
    sine => sin
  );

-- First LPF (255 coefficient FIR, fs=48KHz, fc=4KHz)
-- I-channel
lpf_I : fir1
                port map (
                        clk => clk,
                        rfd => open,
                        rdy => open,
                        din => lpf_I_din,
                        dout => lpf_I_dout
                );
-- Q-channel
lpf_Q : fir1
                port map (
                        clk => clk,
                        rfd => open,
                        rdy => open,
                        din => lpf_Q_din,
                        dout => lpf_Q_dout
                );

-- Second complex multiplier
Inst_comp_mult2 : comp_mult1
                port map (
                        ar => I_in2,
                        ai => Q_in2,
                        br => cos2,
                        bi => sin2,
                        clk => clk_48KHz,
                        pr => cmult_Iout2,
                        pi => cmult_Qout2
                );

-- Second NCO (fixed upshift at 12KHz)
Inst_nco2 : nco2
  PORT MAP (
    clk => clk_48KHz,
    cosine => cos2,
    sine => sin2
  );

-- Realizer circuit
```

```
adder_proc:process(clk_48KHz)
    begin
        if rising_edge(clk_48KHz) then
            adder_out <= adder_I_in + adder_Q_in;
        end if;
    end process;

-- Squarer circuit (AM Demodulator)
Inst_mult : mult
  PORT MAP (
    clk => clk_48KHz,
    a => squarer_in,
    b => squarer_in,
    p => squarer_out
  );

-- Output LPF (255 coefficient FIR, fs=48KHz, fc=4KHz)
lpf_out : fir1
        port map (
            clk => clk,
            rfd => open,
            rdy => open,
            din => lpf_out_din,
            dout => lpf_out_dout
        );

end Behavioral;
```

C7. Auxiliary 16-bit DAC (dac16) -not used

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity dac16 is
Port ( Clk : in STD_LOGIC;
    Data : in STD_LOGIC_VECTOR (15 downto 0);
    PulseStream : out STD_LOGIC);
end dac16;
architecture Behavioral of dac16 is
signal sum : STD_LOGIC_VECTOR (16 downto 0);
begin
PulseStream <= sum(16);
process (clk, sum)
begin
    if rising_edge(Clk) then
        sum <= ("0" & sum(15 downto 0)) + ("0" & data);
    end if;
```

```
end process;
end Behavioral;
```

RESUME

Ali Ibrahim Khalifa HANDER was born in Tripoli / Libya in 1981 and he graduated first and elementary education in this city. He completed high school education in Souq Al-Khamis High School, after that, he started higher diploma program in High Institute for Comprehensive professions - Souq Al-Khamis / Tripoli Department of Electrical and Electronic Engineering in 2000. Then in 2006, he started assignment as a Research Assistant in same High Institute. To complete MSc education, he moved to Karabük University.

CONTACT INFORMATION

Address : High Institute for Comprehensive professions - Souq Al-Khamis / Tripoli Department of Electrical and Electronic Engineering Engineering Campus / Souq Al-Khamis / Tripoli

E-mail : hander_ly@yahoo.com / aliabumalk@gmail.com